Angular 8 e Firebase

Construindo uma aplicação integrada com a plataforma do Google

Casa do Código

KHERONN KHENNEDY MACHADO

Sumário

- ISBN
- Agradecimentos
- Sobre o autor
- Prefácio
- 1. Introdução
- 2. Ambiente de desenvolvimento
- 3. Firebase - A plataforma de serviços do Google
- 4. Serviços
- 5. Componentes - Requisito Login
- 6. Formulários reativos e Pipe - Departamento e Funcionário
- 7. Mais componentes - Requisito Gerenciar Requisições
- 8. Firebase Cloud Storage - Salvando arquivos estáticos
- 9. Firebase Cloud Functions - Criação de usuário e envio de emails
- 10. Deploy da aplicação e considerações finais

ISBN

Impresso e PDF: 978-85-7254-036-0

EPUB: 978-85-7254-037-7

MOBI: 978-85-7254-038-4

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Morar em uma cidade com menos de 20 mil habitantes (Wenceslau Braz - Paraná) e longe de um grande centro é desafiador para o desenvolvedor manter-se atualizado, mas felizmente com a internet e iniciativas como a Casa do Código esses obstáculos estão cada vez menores.

Trabalho com ensino há 15 anos e tenho convicção de que estou no caminho certo. Poder ajudar as pessoas a alcançar seus objetivos profissionais é uma realização que não se traduz em números, mas em sentimentos.

Escrever um livro não é uma tarefa simples. Envolve planejamento, paciência e dedicação do autor e principalmente da família dele.

Assim, inicio agradecendo a Deus, meus pais (Sebastião Kheronn e Dulcinéia) e irmãos (Khauffmann, Khellemann e Khevvellynn) que me apoiaram desde o começo do processo. Minha esposa Cleice pela complacência dos finais de semana. Meu filho Cael, por ser um bebê tão bonzinho e amado.

Meus colegas de trabalho da CRTE no Núcleo Regional de Educação que sempre me incentivam.

A Vivian da Casa Do Código que me orientou na construção com feedbacks sempre assertivos.

Por fim, dedico esse livro a duas pessoas fundamentais na minha vida. Minha vó Bernardina, que não está mais entre nós, mas me ensinou com seu amor a existência de alguém celeste. E finalmente meu primogênito Khaike. Sempre que eu o vejo agradeço a Deus por ter me dado o melhor companheiro do mundo. Papai te ama muito!

Ao leitor, espero que ajude e que possa efetivamente contribuir para seu aprendizado, observando que estudar linguagens, frameworks e

tecnologias é um ato contínuo de busca e construção do conhecimento.

Portanto, esse pode ser só o começo. Jamais o fim!!!

Sobre o autor

Figura -1.1: Kheronn Khennedy Machado

Meu nome é Kheronn Khennedy Machado. Possuo formação em Processamento de Dados pela Fatec Ourinhos e Mestrado em Informática pela Universidade Federal do Paraná.

Iniciei na programação desenvolvendo em Java, Struts, JSF, Spring, mas em 2016 comecei a olhar o JavaScript com outros olhos.

Precisei desenvolver um aplicativo na época e fiquei surpreso com a velocidade do desenvolvimento e com as coisas que eu podia implementar (na minha ignorância, se resumia a *popups*).

Assim, iniciei meus estudos nessa perspectiva e hoje desenvolvo em Angular, Ionic, Cordova e Android.

Sempre que posso, escrevo tutoriais e roteiros que possam ajudar pessoas a iniciarem na programação usando esse framework incrível que é o Angular.

Atualmente, atuo como professor na rede estadual de educação do Paraná e assessor pedagógico em Tecnologias Educacionais no Núcleo Regional de Educação de Wenceslau Braz.

Prefácio

No desenvolvimento de aplicações web e mobile, há disponível uma quantidade expressiva de linguagens, frameworks e ferramentas. Nessa imensidão, é comum o desenvolvedor iniciante ficar perdido e até inseguro sobre qual o melhor caminho para a construção neste segmento.

Para o desenvolvedor front-end, a tarefa é mais complicada, alinhada ao que ele exatamente precisa, seja um formulário, uma SPA (*Single Page Application*), apenas para citar alguns.

O **Angular** (https://angular.io/docs) é uma plataforma que facilita a construção de aplicativos, combinando templates, injeção de dependências, integrado às melhores práticas de desenvolvimento. Principalmente, aplicações responsivas que executem na web, em dispositivos móveis e desktop.

Porém, como nem tudo são flores, codificar com Angular será mais tranquilo para quem possui familiaridade com JavaScript, HTML e CSS. Outro aspecto que facilita é possuir algum conhecimento em linguagens orientadas a objetos.

O objetivo desta obra é apresentar as principais características da plataforma, utilizando a **versão 8**, lançamento mais recente do framework, através da implementação de uma aplicação que guiará os capítulos. Logo, você não vai encontrar um capítulo teórico dedicado aos conceitos de componentes, serviços, roteamentos ou validação de formulários, porém, vai, sim, explorar esses tópicos identificados dentro de um requisito do projeto.

Ainda, vamos integrar a aplicação ao **Firebase** do Google, utilizando diversos recursos como banco de dados, autenticação, armazenamento de arquivos, execução de funções no lado do servidor e hospedagem do sistema.

Público-alvo e pré-requisitos

Este é livro é destinado a todos que desejam construir aplicações JavaScript com alta produtividade e usar os principais recursos do Firebase no desenvolvimento de soluções escaláveis sem se preocupar com o gerenciamento da infraestrutura.

Como pré-requisito é necessário que o/a leitor/a tenha conhecimentos básicos em HTML, CSS e JavaScript. O foco será nas particularidades do framework na implementação dos requisitos, explorando as potencialidades e poder do Angular.

Todo código desenvolvido durante os capítulos estará disponível no repositório do GitHub, indicado nos finais das seções ou durante os capítulos.

Ao final do livro, você terá desenvolvido um sistema de requisições completo, explorando os principais conceitos do framework, além de integrar a aplicação aos serviços da plataforma Firebase.

Capítulo 1
Introdução

Neste capítulo, vamos apresentar brevemente as tecnologias envolvidas na codificação da aplicação e o estudo de caso que guiará a implementação em Angular 8.

1.1 Angular

Angular é um framework mantido pelo Google para a construção de aplicações web, mobile e desktop.

Lançado em 2012 como AngularJS, tornou-se um dos frameworks JavaScript mais populares, simplificando a forma de programar para web, alinhando componentes a padrões de projeto como injeção de dependências e arquitetura MVC (Model-View-Controller).

A partir da segunda versão o Angular foi totalmente restruturado. No momento da escrita deste livro (maio de 2019), a versão utilizada é a 8.

> **As versões**
>
> Não se preocupe com a versões do Angular. O Google reconhece o compromisso de estabilidade da estrutura, garantindo que ferramentas e práticas não se tornem obsoletos.

Uma das principais vantagens na utilização de frameworks é forçar o desenvolvedor a adoção de padrões. Embora isso seja motivo de discussões, pois de certa forma leva o desenvolvimento numa caixinha, por outro, traz grandes vantagens como produtividade.

Sobre o tema produtividade temos especificamente o **Angular CLI**. Essa ferramenta de comando permite a construção de aplicações e a geração de artefatos, componentes e classes de forma rápida. Faremos uso intenso dessa ferramenta durante a construção da aplicação.

Dentre outros atributos que pesam a favor do desenvolvimento utilizando Angular, temos recursos modernos de plataforma da web para fornecer experiências semelhantes a aplicativos com PWA (*Progressive Web Application*). Com esses recursos e diversos produtos é possível criar aplicativos móveis utilizando Cordova, Ionic e Native script ou desenvolver sistemas desktops, com acesso a APIs dos sistemas operacionais.

Logo, nota-se a versatilidade do framework na construção de soluções para diversas plataformas e dispositivos.

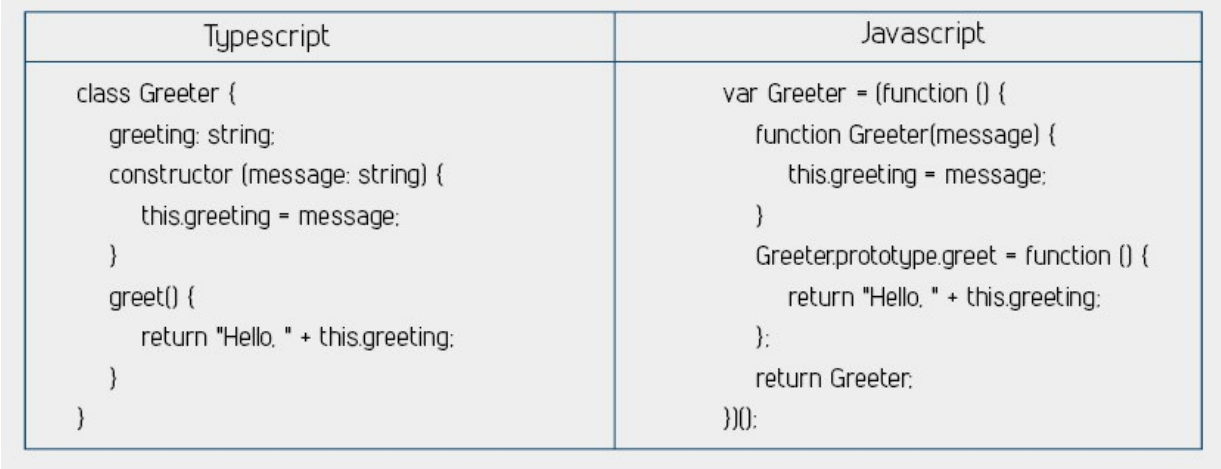
Angular é atualmente um dos grandes frameworks do mercado, ao lado de React e Vue e com isso é crescente a demanda por profissionais habilitados na construção de soluções de software.

Como linguagem *background*, o Angular utiliza o TypeScript, que veremos a seguir.

1.2 TypeScript

É a linguagem utilizada para codificação no Angular, uma sintaxe clara é fácil de entender compilado para JavaScript. Criada e mantida pela Microsoft (que também disponibilizou a IDE Visual Studio Code), ela funciona como um superconjunto do ES2015 que fornece entre outros benefícios a tipagem estática, classes, módulos e namespaces.

Na sequência, um exemplo de classe em Type e JavaScript:



Typescript	Javascript

class Greeter {
greeting: string;
constructor (message: string) {
this.greeting = message;
}
greet() {
return "Hello, " + this.greeting;
}
}
	var Greeter = (function () {
	function Greeter(message) {
	this.greeting = message;
	}
	Greeter.prototype.greet = function () {
	return "Hello, " + this.greeting;
	};
	return Greeter;
	})();

Figura 1.1: Classes em Type e Javascript. Fonte: https://i1.wp.com/www.dunebook.com/wp-content/uploads/2017/03/Typescript-and-the-Javascript-code.jpg?ssl=1

Caso você tenha interesse em conhecer melhor sobre TypeScript recomendo os links a seguir sobre a sintaxe:

- https://www.alura.com.br/curso-online-typescript-parte1 - TypeScript parte 1: Evoluindo seu JavaScript. Curso em português da Alura, voltado ao básico, apresenta desde vantagens da tipagem a conceitos de herança e reaproveitamento do código.
- https://www.typescriptlang.org/docs/home.html - Documentação oficial do TypeScript. Site em inglês, há no menu um playground onde é possível criar códigos e acompanhar as saídas e a compilação em JavaScript.

Não há necessidade de conhecimento intermediário ou avançado na linguagem. As funções serão explicadas e toda vez que um recurso novo surgir será fornecido detalhes da sua implementação.

1.3 O estudo de caso

É muito frustrante quando procuramos material de estudos em uma linguagem ou tecnologia nova e não encontramos nada “real”. A maioria são exemplos *toys* que até contribuem para o entendimento de uma abordagem, mas que não trazem significados concretos quando construímos sistemas.

Neste livro, proponho um sistema que permita explorar os conceitos fundamentais do framework como componentes, rotas e serviços. Vamos integrar nossa aplicação com o Firebase, utilizando serviços de autenticação, armazenamento de dados e execução de funções no lado do servidor. Embora de contexto simples, cobrirá todos esses aspectos, e a partir disso você poderá construir desde sistemas com modelagem simples aos mais complexos.

No macrocontexto, vamos criar um sistema de requisições internas. A aplicação deve permitir o gerenciamento dessas requisições oriundas de qualquer funcionário, devidamente cadastrado, e a solicitação deve ser destinada a um departamento. Somente funcionários do departamento solicitado é que podem atualizar o status da requisição.

A imagem a seguir apresenta os requisitos funcionais.

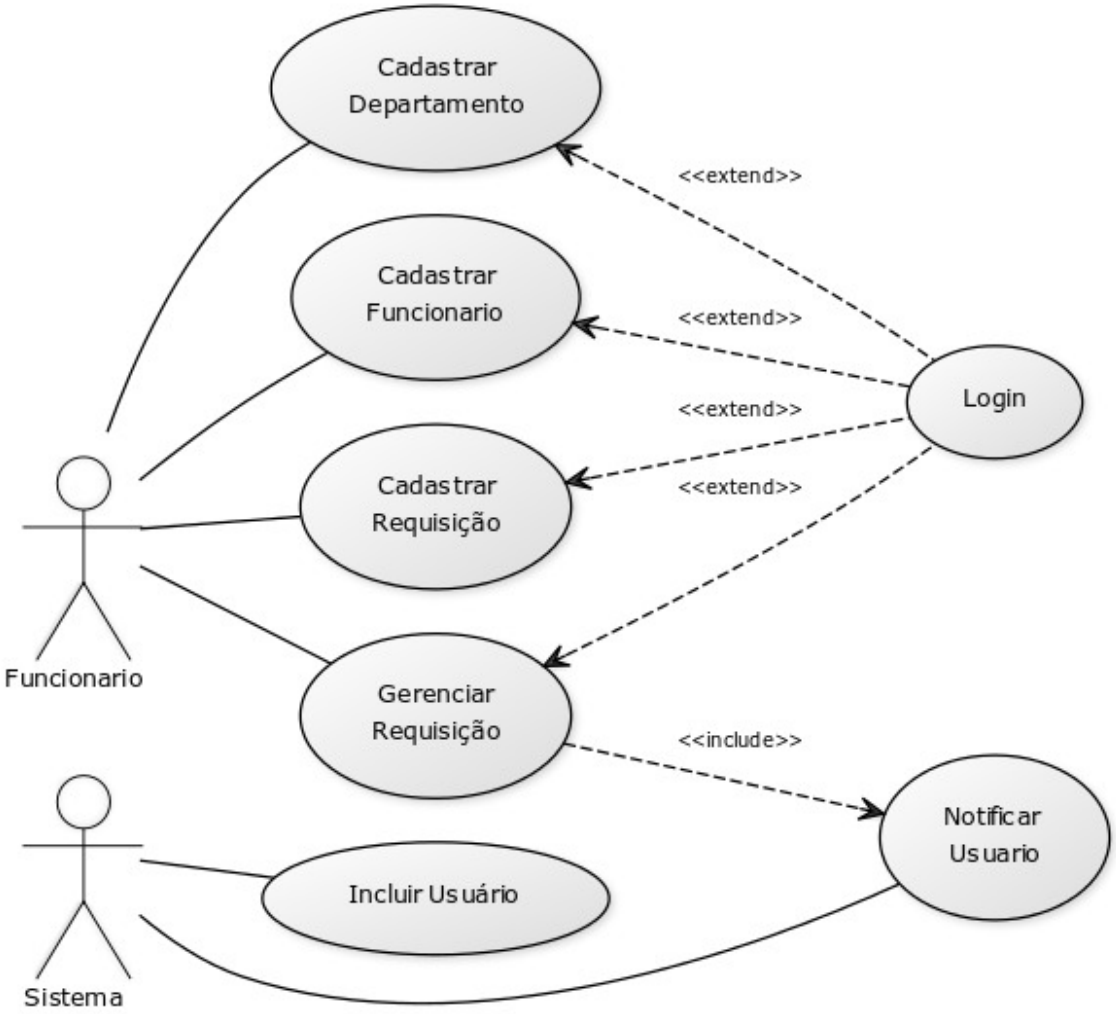


Figura 1.2: Use Case para o sistema de requisições

A cada atualização do status, o solicitante será notificado por e-mail sobre o andamento da requisição. Deste modo, podemos trabalhar com funções no servidor, mais precisamente com o **Firebase Functions**, representado pelo *user* Sistema na figura.

Com este caso de exemplo, será possível lidar com uma grande quantidade de características do framework, e nos guiaremos sempre pelas práticas recomendadas na construção de sistemas.

No próximo capítulo vamos preparar o ambiente, instalando as dependências necessárias para início da codificação.

CAPÍTULO 2
Ambiente de desenvolvimento

Quem está acostumado com a instalação de programas com interfaces gráficas, o famoso *"Clique em 'próximo'"*, pode estranhar um pouco o uso de linhas de comandos em terminais. No entanto, com o uso frequente, nota-se a praticidade na criação, testes e publicação de projetos. Somente no primeiro programa a instalação é gráfica e necessária para passos posteriores.

2.1 NodeJS

O Node.js inclui o gerenciador de pacotes `npm`, responsável pela instalação de plugins, bibliotecas e o próprio Angular. No site https://nodejs.org/en/download, encontre a versão correspondente ao seu sistema operacional. A instalação é simples e não requer configuração opcional.

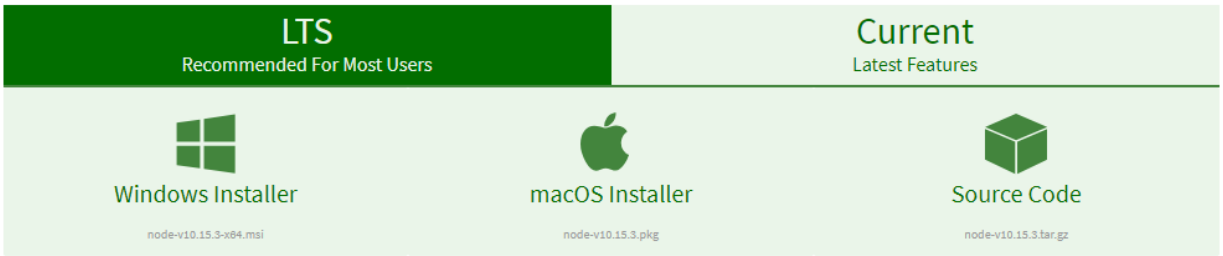


Figura 2.1: Opções de download NodeJS

Após a instalação, abra um terminal (`cmd.exe` ou PowerShell no Windows | terminal no Mac) e digite o comando:

```
node -v
```

Se tudo ocorreu bem a versão do programa é impressa na tela. Depois de instalado, o comando `npm` ficará disponível no terminal. A

instalação de novos pacotes segue o formato ***npm install nome-do-pacote***.

Para os próximos programas utilizaremos esse comando para efetivar as instalações.

2.2 Instalação do Angular 8

Vamos instalar o framework Angular na versão 8 (8.0.1) utilizando o gerenciador de pacotes NPM. Com o terminal aberto, digite:

```
npm install -g @angular/cli
```

No Linux ou OSX terminal acrescente o `sudo` :

```
sudo npm install -g @angular/cli
```

Após a conclusão do download temos acesso global ao `Angular CLI` através do comando `ng` seguido de outros parâmetros. Ainda no terminal, podemos conferir a versão instalada informando `ng --version` . A imagem posterior apresenta o resultado do comando.

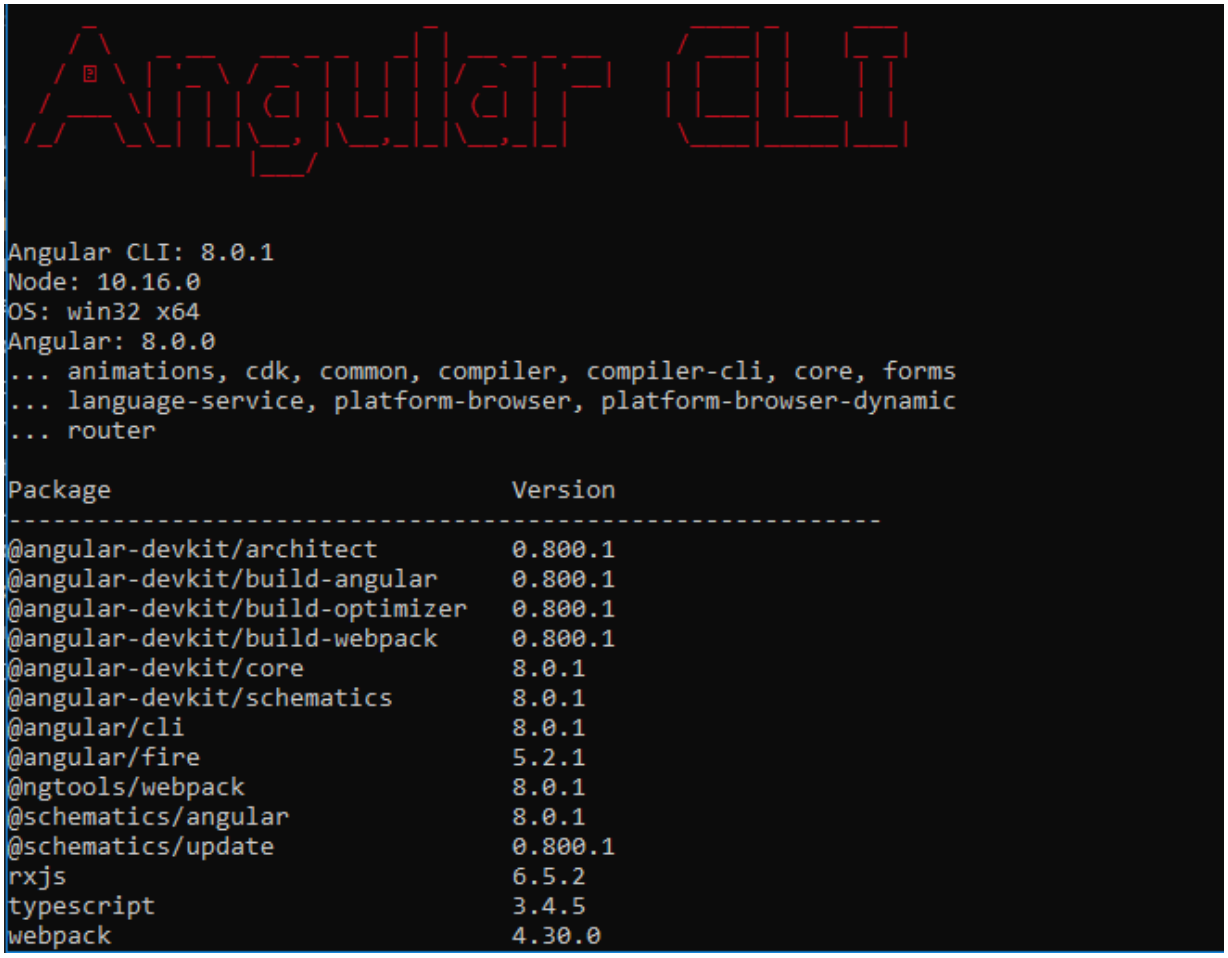


Figura 2.2: Resultado do comando ng --version

Entre outras informações, o console apresenta a versão do Angular CLI (8.0.1), no momento da escrita do livro, versão no Node e dos pacotes que compõem o `core` do framework.

2.3 Editor de desenvolvimento

Esta seção apresenta uma sugestão de editor para codificar seus projetos.

O Visual Studio Code, ou popularmente chamado de VSCode, está disponível em https://code.visualstudio.com. Além de possuir uma

quantidade enorme de extensões que facilitam a programação, como *autocomplete*, indentação, integração nativa com GIT, entre outros (a lista é extensa...), a ferramenta é leve e possui um terminal integrado.

Baixe e instale o editor.

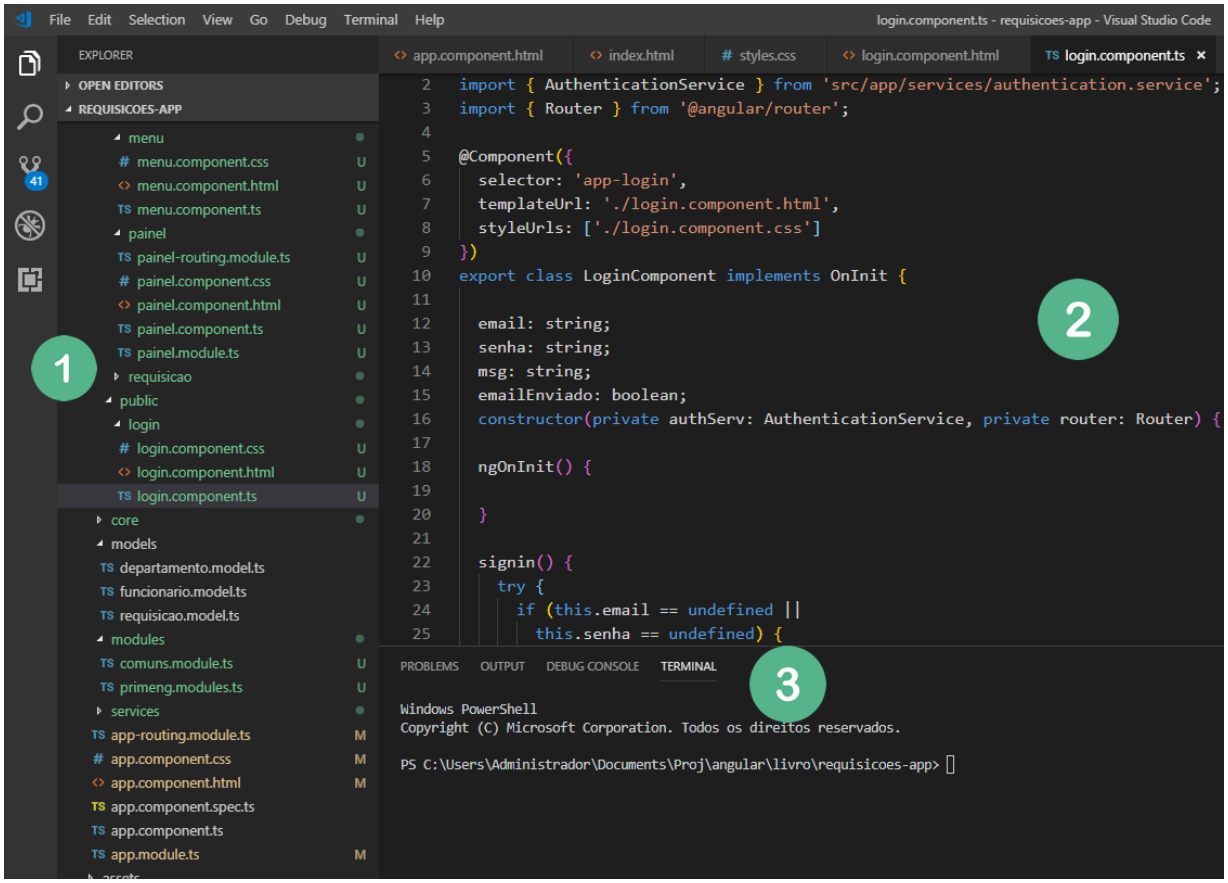


Figura 2.3: Visual Studio Code

(1) Menu lateral- Estrutura do projeto, arquivos e pastas;
(2) Área central - Arquivos abertos em abas;
(3) Acessórios - Abas que apresentam os problemas, saídas e um terminal integrado;

Nosso ambiente para codificar está pronto. Na próxima seção vamos criar o projeto.

2.4 Angular CLI - Criação do projeto

Utilizando o prompt de comandos, vamos utilizar o `Angular CLI` para a criação do projeto e geração de outros artefatos. Abra o terminal e navegue até um diretório de sua escolha e digite:

```
ng new requisicoes-app
```

Vamos entender os comandos:

```
* ng - CLI do angular
* new - criação de um novo projeto
* requisicoes-app - nome do projeto
```

Informamos o mínimo necessário para iniciar um novo projeto. Há outros parâmetros que você pode passar na construção de um novo projeto.

A relação de todos os comandos pode ser encontrada na página oficial do Angular CLI wiki (https://github.com/angular/angular-cli/wiki). Após confirmarmos o comando com a tecla `Enter`, algumas questões serão exibidas no terminal:

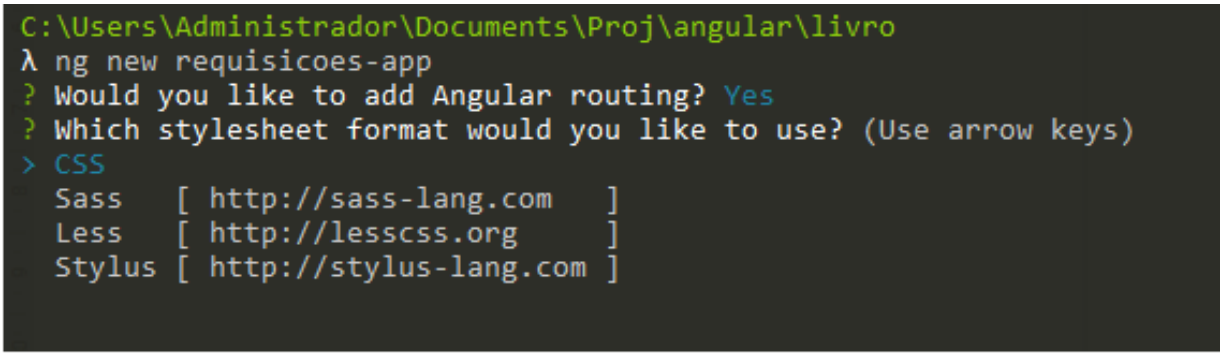

C:\Users\Administrador\Documents\Proj\angular\livro
λ ng new requisicoes-app
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? (Use arrow keys)
> CSS
Sass [http://sass-lang.com]
Less [http://lesscss.org]
Stylus [http://stylus-lang.com]

Figura 2.4: Criando o projeto com Angular CLI

A primeira diz respeito às rotas. Veremos no capítulo 5 o que são e como mapear.
Neste momento, o importante é confirmar para que crie um arquivo específico que mapeia as rotas. Informe Y.

A segunda é sobre o formato do estilo. Deixaremos o primeiro, CSS, pois não utilizaremos recursos avançados que justifiquem a escolha de outras linguagens como SASS ou SCSS.

Será criada toda estrutura de arquivos e serão baixadas as dependências iniciais da aplicação, como os pacotes `common`, `core` e `compiler`. Detalhes dos pacotes são encontrados no arquivo `package.json`, na raiz do projeto.

Vamos navegar até o diretório do projeto criado para executar o projeto. Para isso, no terminal insira:

```
cd requisicoes-app
```

Já é possível ver o projeto executando com o comando:

```
ng serve --o
```

O argumento `--o` indica para abrir no navegador padrão o endereço: http://localhost:4200/.

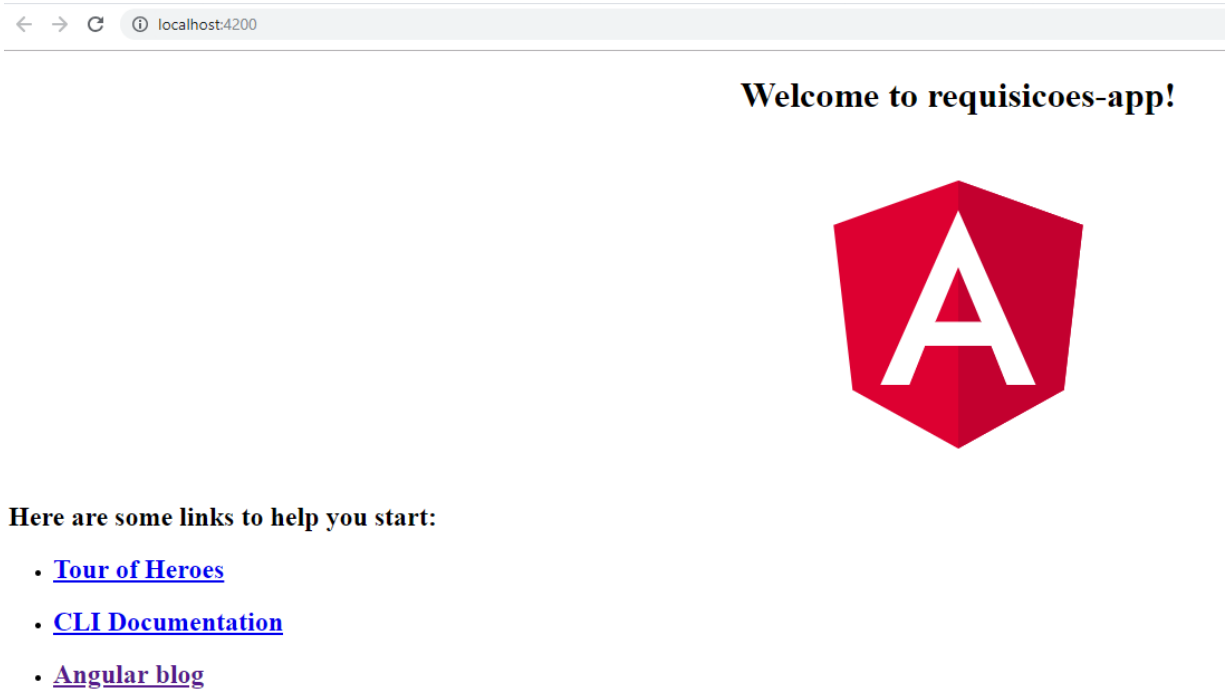


Figura 2.5: Servidor executando localmente o projeto

O Angular inicia um servidor local com seu projeto compilado e executado.

2.5 Arquitetura da aplicação

Vamos conferir a estrutura criada pelo Angular e apresentar a arquitetura gerada pelo `Angular CLI`. Faremos isso utilizando o VSCode.

Na seção anterior, o último comando estava executando o projeto. Interrompa o processo com *CTRL + C* e logo em seguida informe:

```
code .
```

Este comando abre o editor com o projeto criado. A figura seguinte apresenta a visão lateral de pastas e arquivos.

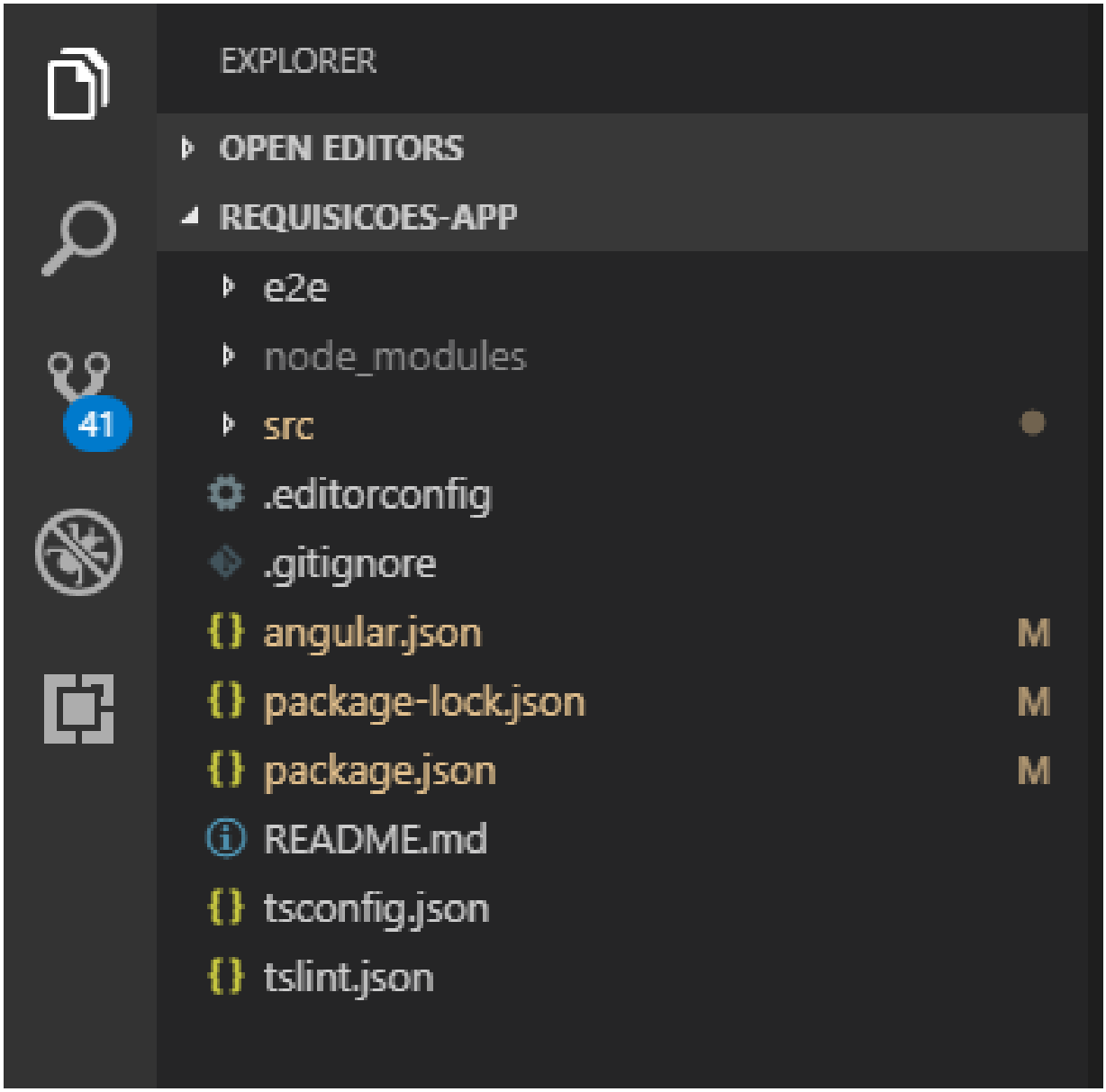


Figura 2.6: Estrutura inicial do projeto

No primeiro nível do projeto podemos destacar as seguintes pastas e arquivos:

```
* e2e: contém arquivos responsáveis pelos testes unitários;
* node_modules: armazena todas as bibliotecas listadas no arquivo package.json;
* package.json: referências de todas as dependências npm que o projeto utiliza;
* angular.json: configurações do Angular CLI para criação, execução e
```

testes;
* src: pasta contendo arquivos-fontes do projeto.

Na pasta `src` podemos destacar os seguintes arquivos:

* index.html: arquivo root, a partir dele é executada a SPA;
* app.components.ts: nomeado como AppComponent, define a lógica para o componente raiz;
* app.component.html: define o template HTML associado ao componente AppComponent;
* app.component.css: define a base CSS;
* app.module.ts: define o módulo raiz, chamado de AppModule; é o local onde declaramos outros módulos, componentes e serviços;
* assets: local onde salvamos arquivos estáticos da aplicação como imagens.

2.6 Estilizando a aplicação com Bootstrap

Planejar a fonte, tamanho, disposição, cor, layouts e temas são tarefas com que o desenvolvedor front-end lida no processo de construção de aplicações.

Uma possibilidade é a utilização de um framework que forneça componentes estilizados e adaptativos. Nesse sentido, optei pelo uso do `Bootstrap` já personalizado com temas, disponível no site https://bootswatch.com.

> **Bootstrap**
>
> Bootstrap é um framework web com código-fonte aberto para desenvolvimento de componentes de interface e front-end para sites e aplicações web usando HTML, CSS e JavaScript, baseado em modelos de design para a tipografia, melhorando a experiência do usuário em um site amigável e responsivo. (https://pt.wikipedia.org/wiki/Bootstrap_(framework_front-end))

Após escolher um tema do seu gosto (eu escolhi o tema Materia), copie o conteúdo do arquivo `bootstrap.min.css` para arquivo `style.css` do projeto.

O próximo passo é planejar a organização dos artefatos de nossa aplicação, de acordo com sua funcionalidade. A imagem a seguir ilustra a composição das pastas que deixaremos criado abaixo de `src/app` .

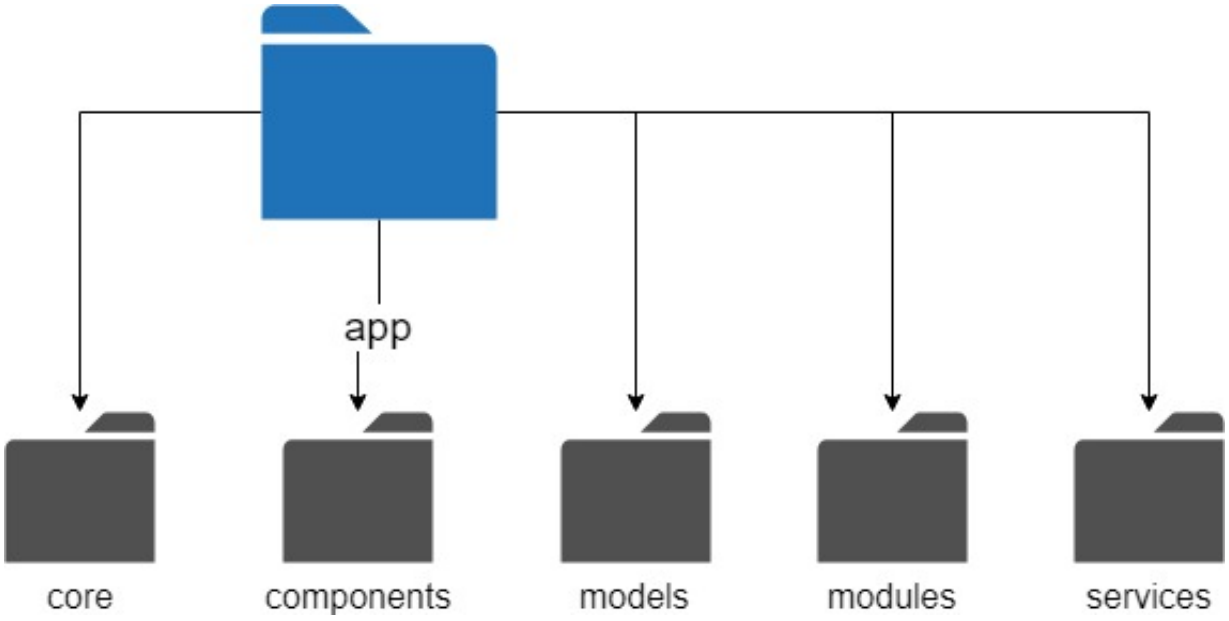


Figura 2.7: Paleta do terminal

Na pasta `core` serão colocadas as classes e interfaces disponíveis para toda a aplicação.

Na pasta `components` ficarão os componentes. Cada componente é composto de uma classe, um template e um arquivo de estilo.

Os modelos serão inseridos em `models` . São as classes que representam as entidades do negócio. Em `modules` ficarão os módulos de componentes, que serão reutilizados.

Finalmente, a pasta `services` será o destino de classes que acessam o banco, autenticação e outros recursos.

2.7 PrimeNG - Coleção de componentes ricos

No processo de desenvolvimento de aplicações é comum a necessidade de componentes específicos para determinadas tarefas, por exemplo para criar tabelas, caixas de entradas personalizadas e dialogs. Codificar do zero pode levar tempo e recursos.

Dessa forma, podemos integrar nossa aplicação para finalizar com a biblioteca PrimeNG.

> **PRIMENG**
>
> PrimeNG é uma coleção com mais de 80 componentes de UI ricos para Angular. Todos os widgets são open source e gratuitos para uso sob licença MIT.

No site da biblioteca, https://www.primefaces.org/primeng, temos acesso à documentação com exemplos de utilização e demonstração de cada componente.

A seguir, um exemplo dos passos para utilizar um componente que exibe um calendário.

1. Realize o `import` do módulo específico:

```
import {CalendarModule} from 'primeng/calendar';
```

2. Declaração da tag do componente no html:

```
<p-calendar [(ngModel)]="value"></p-calendar>
```

A figura a seguir apresenta o componente.

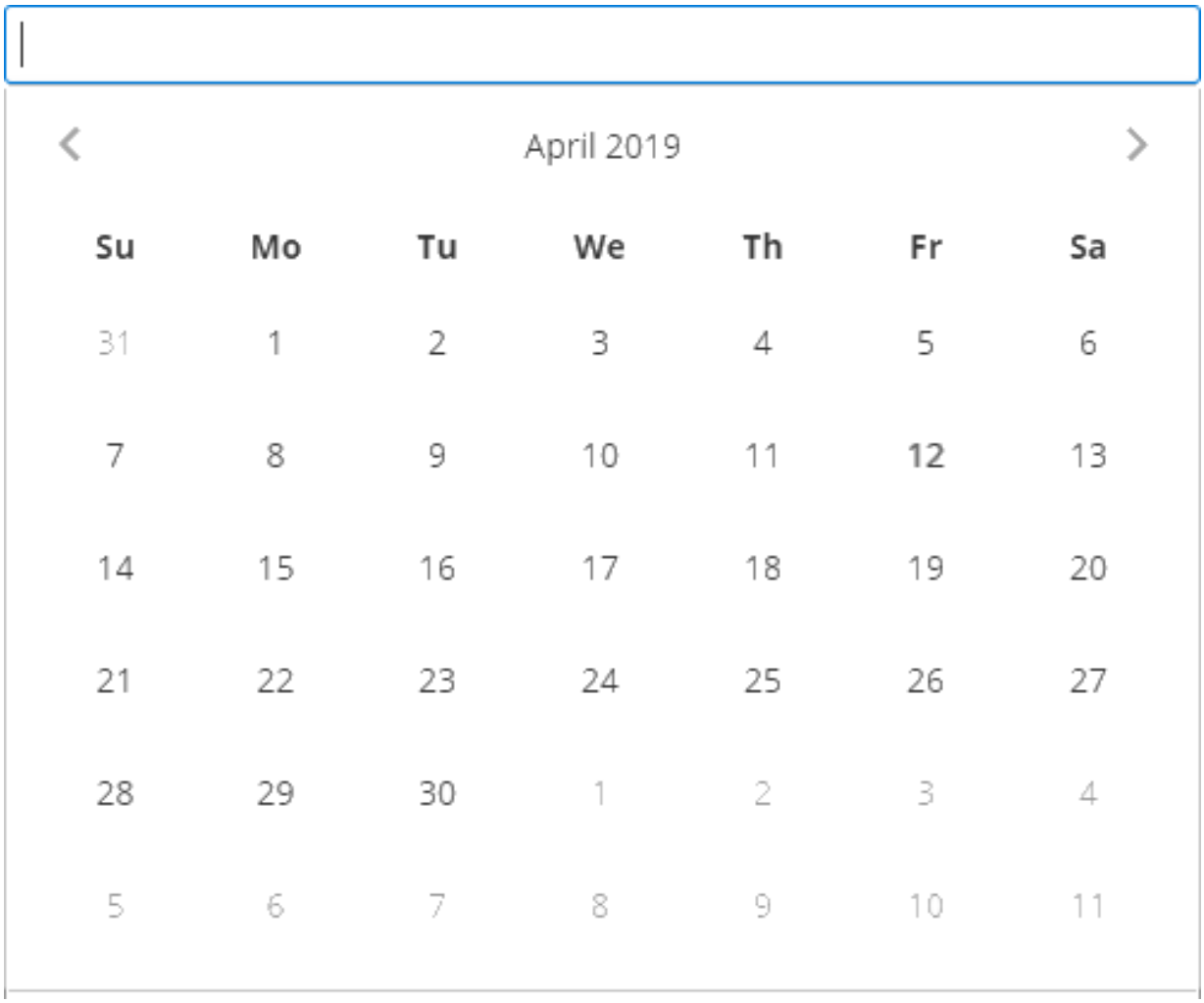


Figura 2.8: Componente calendário

Para instalar a biblioteca, no terminal digite os seguintes comandos:

```
npm install primeng --save
npm install primeicons --save
npm install @angular/animations --save
```

Com isso, instalamos a biblioteca *PrimeNG*, um conjunto de ícones e uma *lib* para habilitar animações. Essa última requer a importação do módulo, passo que faremos a seguir.

MÓDULOS

São arquivos definidos com a diretiva *NGModule* para agrupar componentes, diretivas e serviços.

No arquivo `app.module.ts` disponível na pasta `src/app` vamos realizar o `import` do módulo, declarando no array de imports, conforme trecho do código a seguir:

```
import {BrowserModule} from '@angular/platform-browser';
import {BrowserAnimationsModule} from '@angular/platform-
browser/animations';

@NgModule({
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        //...
    ],
    //...
})
],
```

O próximo passo é definir no arquivo `angular.json` , encontrado na raiz do projeto, os caminhos dos arquivos de estilos. Encontre a seção *styles* e insira 3 linhas abaixo de `src/styles.css` , ficando assim:

```
"styles": [
  "src/styles.css",
  "node_modules/primeicons/primeicons.css",
  "node_modules/primeng/resources/themes/nova-light/theme.css",
  "node_modules/primeng/resources/primeng.min.css",
],
```

Definimos os arquivos de estilos, ícones e o tema. Você pode escolher outros consultando a página da biblioteca.

A partir de agora, para cada componente que usarmos, devemos importar o módulo correspondente, conforme exemplo no começo da seção.

Nosso ambiente e projeto já estão devidamente configurados e temos tudo pronto para iniciarmos a implementação.

No próximo capítulo vamos criar uma conta no `Firebase` para encaminhar as questões de armazenamento e autenticação.

Capítulo 3
Firebase - A plataforma de serviços do Google

Um dos principais dilemas na construção de qualquer software está na criação, organização e manutenção na base de dados. Para dispositivos, essa funcionalidade deve levar em consideração ainda outros aspectos na implementação como disponibilidade, tempo da consulta, formas de acesso e armazenamento, para citar algumas.

Levando em conta este cenário, o Google oferece uma infraestrutura recheada de recursos, possibilitando qualidade e escalabilidade. O desenvolvedor pode direcionar seus esforços a outras atividades, como no engajamento dos usuários, deixando que o Google dimensione automaticamente a infraestrutura.

Uma questão interessante está no preço do serviço. O plano *SPARK* é gratuito e possui limites generosos, permitindo ao desenvolvedor testar e publicar as principais funcionalidades sem custos.

Neste capítulo realizaremos dois passos fundamentais:

1. Criação do projeto na página do Firebase para configuração do nosso provedor de back-end paras os serviços.
2. Instalação das bibliotecas @angular/fire e do SDK.

Na próxima seção criaremos o projeto na página oficial do Firebase e depois, retornaremos ao código para instalar as dependências, integrar e configurar o acesso.

3.1 Criação do projeto no console do Firebase

Acesse o site https://firebase.google.com/?hl=pt-br e clique em `Ir para o console`. Será necessário ter uma conta Google para realizar o

login.

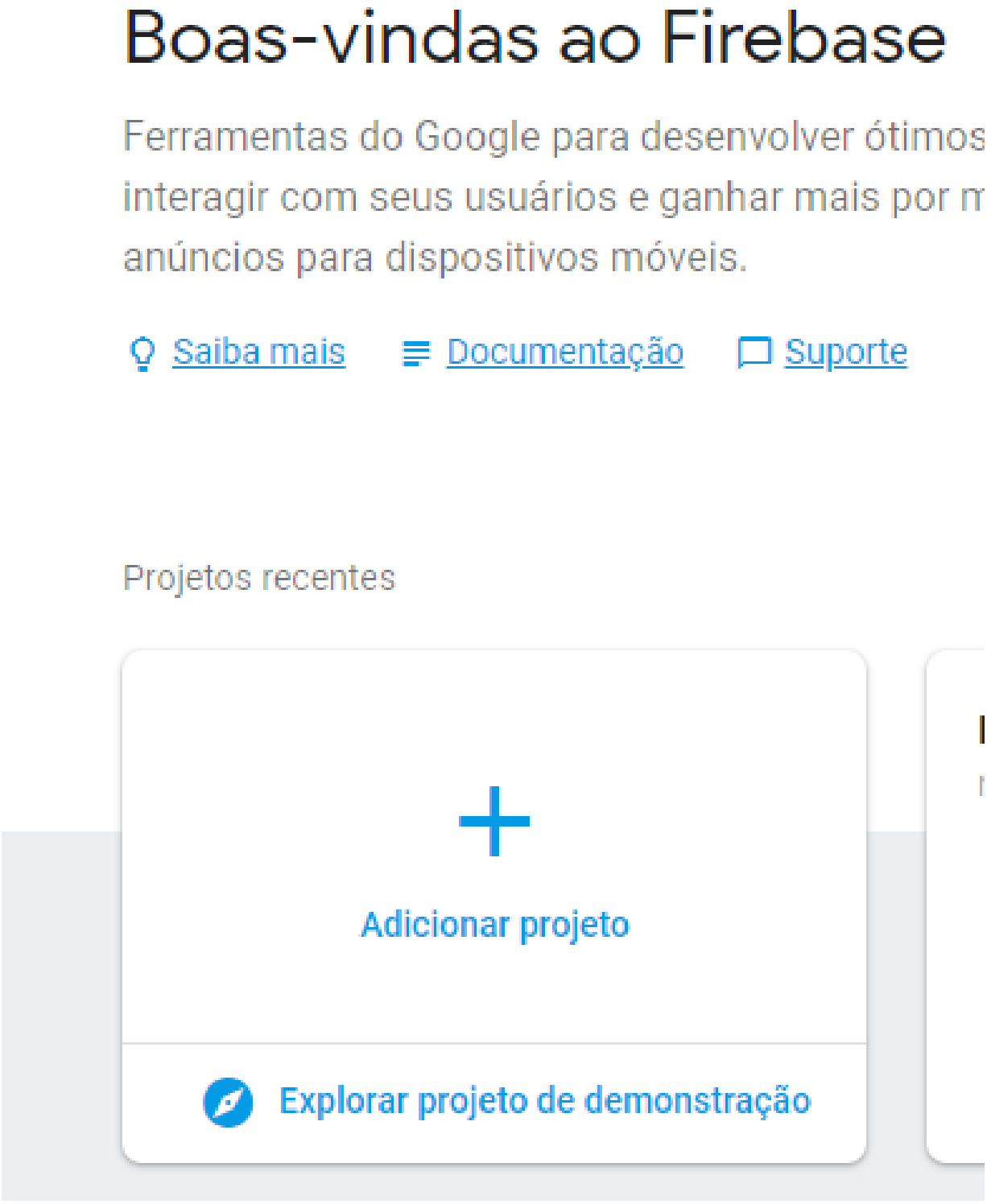


Figura 3.1: Adicionando o projeto

Após informados o usuário e senha, você será redirecionado para a tela do console. Clique no botão `Adicionar projeto`. Dê um nome e selecione o país/região.

Adicionar um projeto

Nome do projeto

requisicoes-app

+ iOS + </>

Dica: os projetos abrangem apps de várias plataformas

Código do projeto

requisicoes-app

Locais

Estados Unidos (Analytics)
nam5 (us-central) (Cloud Firestore)

- [x] Usar as configurações padrão para compartilhar os dados do Google Analytics para Firebase

  - ✓ Compartilhar seus dados do Analytics com todos os recursos do Firebase
  - ✓ Compartilhe seus dados do Analytics com o Google para melhorar os produtos e serviços da empresa
  - ✓ Compartilhe seus dados do Analytics com o Google para ativar o suporte técnico
  - ✓ Compartilhe seus dados do Analytics com o Google para ativar o Comparativo de mercado
  - ✓ Compartilhe seus dados do Analytics com os especialistas em contas do Google

- [x] Aceito os termos do controlador. Isso é necessário ao compartilhar dados do Analytics para melhorar os produtos e serviços do Google. Saiba mais

Cancelar **Criar projeto**

Figura 3.2: Criando o projeto na página do Firebase

Ao clicar no botão `Criar projeto`, aguarde alguns instantes. Haverá um redirecionamento para a tela administrativa, conforme a figura seguinte.

Figura 3.3: Dashboard do Firebase

A partir desse painel temos acesso a diversos serviços e configurações para integrar no projeto.

3.2 Autenticação

Desenvolver um sistema de autenticação seguro pode levar tempo e recursos. O `Firebase Authentication` fornece uma solução completa, proporcionando uma experiência positiva para os usuários finais.

Vamos habilitar a autenticação no painel. Observe na imagem anterior, no menu esquerdo, *Authentication*. Clique para ativar o método e escolher o provedor de login.

Há vários provedores (métodos de login), como Facebook, Twitter, Google. Vamos ativar a primeira opção, `E-mail/Senha`.

Ative e confirme no botão `Salvar`, conforme a imagem seguinte:

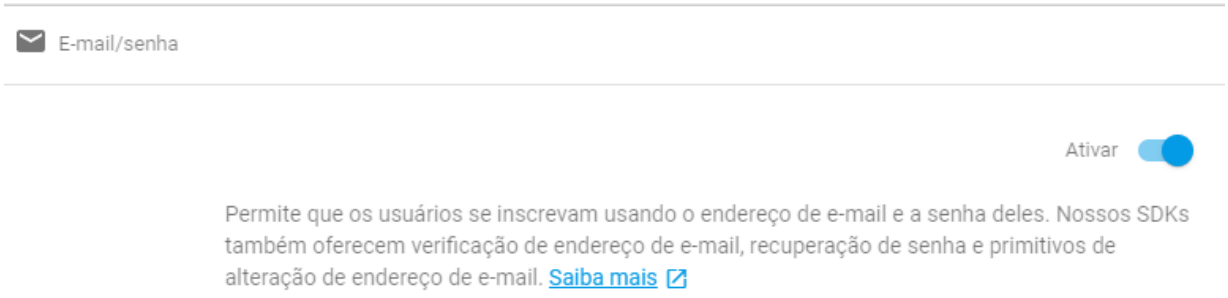

E-mail/senha
Ativar
Permite que os usuários se inscrevam usando o endereço de e-mail e a senha deles. Nossos SDKs também oferecem verificação de endereço de e-mail, recuperação de senha e primitivos de alteração de endereço de e-mail. Saiba mais

Figura 3.4: Habilitante o método de autenticação

Outro passo importante é definir um usuário. Clique na guia `Usuários`.

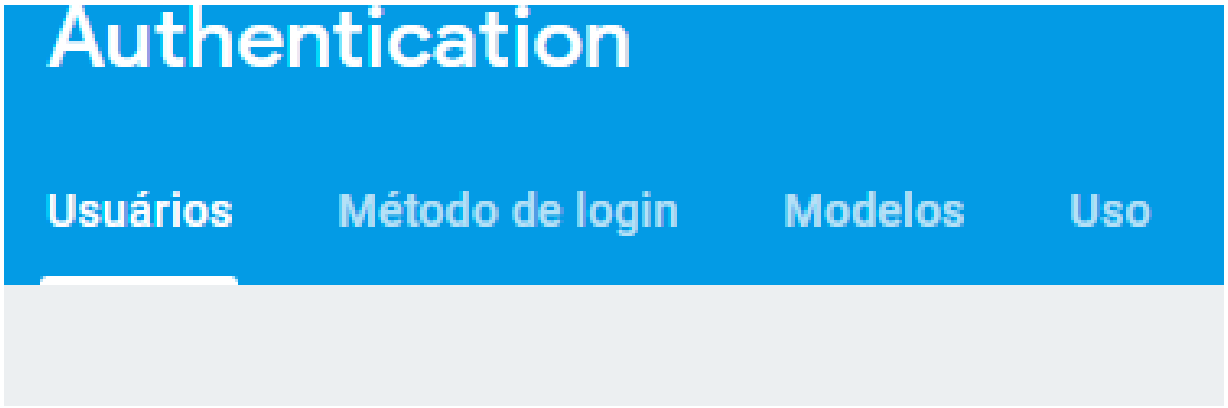


Figura 3.5: Adicionando o usuário

Logo em seguida, clique no botão `Adicionar usuário`.

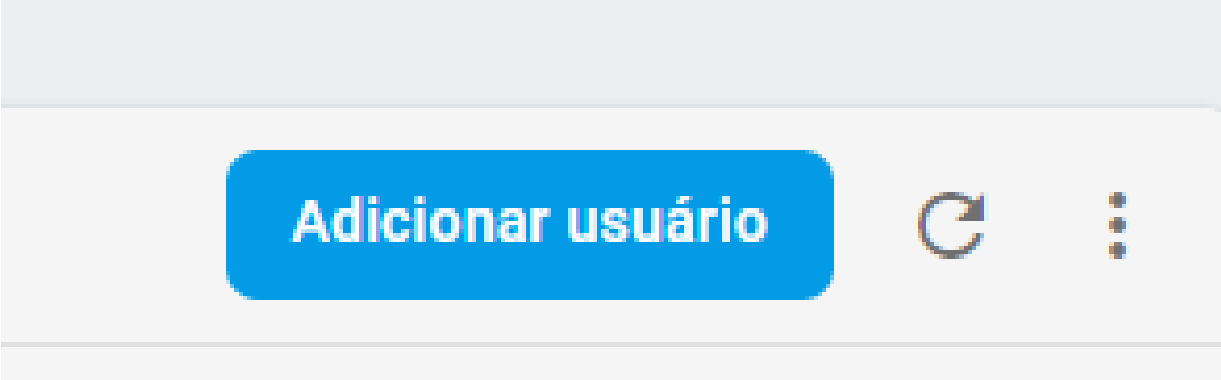


Figura 3.6: Adicionando o usuário

Dê um e-mail e uma senha com no mínimo 6 caracteres. Já temos um usuário para acessar os dados.

Vamos escolher a solução de dados.

3.3 Cloud Firestore - Armazenamento de dados em escala global

Vamos à questão crucial no desenvolvimento de aplicações, que está no armazenamento de dados. O desenvolvedor precisa ater-se à disponibilidade, segurança, manutenção e principalmente à escalabilidade.

O Firebase propõe a resolução dessas questões através do `Cloud Firestore`. É uma solução de banco de dados baseada em documentos NoSQL que facilita a gerência, sincronização e consulta de dados em escala global.

Os dados são estruturados em coleções de documentos, e conforme aumenta a base, as consultas são dimensionadas conforme os resultados, permitindo o escalonamento transparente para a empresa.

No *Dashboard*, clicando no menu esquerdo, `Database`, você verá a seguinte tela na primeira vez.

Figura 3.7: Opção para integração

Ao prosseguir com a criação do banco, uma tela solicita qual modo será selecionado ao iniciar o acesso aos dados.

No `modo bloqueado`, o banco fica privado, assim, todas as leituras de gravação de terceiros serão negadas. Já no `modo de teste`, todas as leituras e gravações são permitidas.

Regras de segurança do Cloud Firestore
Depois de definir sua estrutura de dados, você precisará gravar regras para proteger seus dados.
Saiba mais
Iniciar no modo bloqueado
Para tornar seu banco de dados privado, basta negar todas as leituras e gravações
Iniciar no modo de teste
Configure rapidamente ao permitir todas as leituras e gravações no seu banco de dados
service cloud.firestore {
match /databases/{database}/documents {
match /{document=**} {
allow read, write: if false;
}
}
}
Todas as leituras e gravações de terceiros serão negadas
Ativar o Cloud Firestore impedirá que você use o Cloud Datastore neste projeto, principalmente no aplicativo associado ao App Engine.
Cancelar
Ativar

Figura 3.8: Segurança

Para nossa aplicação, somente usuários autenticados terão acesso (gravação e leitura) aos dados. Portanto, deveremos personalizar as regras de segurança.

Clique em `Ativar`. Aguarde finalizar o processo e clique na guia `Regras`.

Há um campo com a definição escolhida anteriormente. Altere a condição do `if` conforme o código a seguir:

```
match /databases/{database}/documents {
    match /{document=**} {
      allow read, write: if request.auth.uid != null;
    }
`
```

Essa regra determina a leitura e escrita de todos os documentos da base somente se houver um usuário logado. É possível definir

regras personalizadas para cada coleção ou operação de acesso.

Na guia `Dados` podemos incluir coleções e documentos manualmente. A imagem a seguir mostra a inclusão da coleção `funcionarios`.

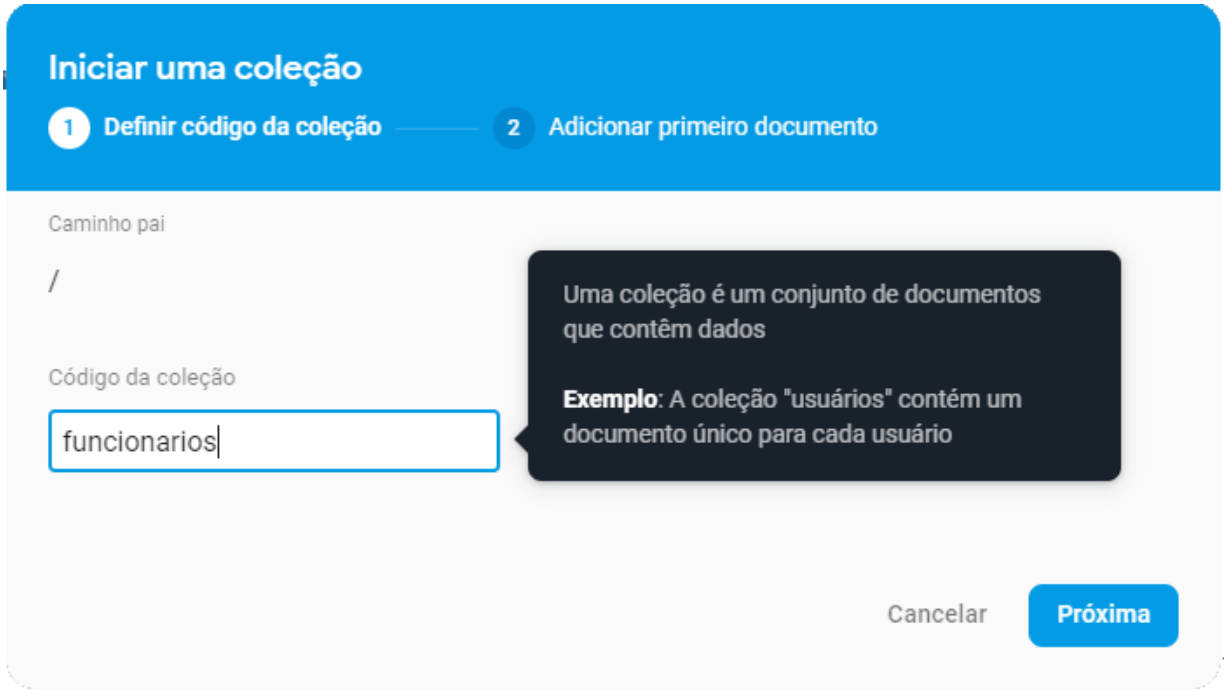


Figura 3.9: Inserindo coleções

Logo em seguida é possível a criação de um documento. No entanto, nossa aplicação é que fará a gestão das informações, portanto, finalizamos o passo do banco de dados.

Antes de voltar ao código, vamos copiar os dados de conexão para utilizar no código da aplicação.

Para isso, clique no menu `Project Overview`. Abaixo do título *Comece adicionando o Firebase ao seu aplicativo*, temos a as opções de integração com as plataformas iOS, Android, Web e Unity.

Comece adicionando o Firebase ao seu aplicativo
Nosso SDK principal tem acesso à maioria dos recursos do Firebase e inclui o Firebase Analytics de apps para iOS e Android
iOS

Figura 3.10: Opção para integração

Clique no terceiro ícone. Abrirá um *dialog* com informações de conexão ao Firebase.

Figura 3.11: Adicionar o Firebase ao aplicativo Web

Vamos copiar esses valores para voltar no projeto Angular.

3.4 @AngularFire - A biblioteca oficial para Firebase e Angular

Voltando ao projeto, vamos trabalhar com a biblioteca oficial que auxilia nas operações com o Firebase e o próprio SDK. No terminal de comandos, digite:

```
npm install @angular/fire firebase  --save
```

Serão adicionadas duas dependências no projeto. No momento em que escrevo o livro, suas versões são:

- @angular/fire@5.1.2
- firebase@5.9.0

Não se preocupe se, ao instalar, você perceber versões maiores. O projeto `@angularfire` está em constante atualização para acompanhar as mudanças do SDK.

Edite o arquivo `/src/environments/environment.ts` e *cole* as propriedades de acesso do Firebase. O código ficará assim:

```
export const environment = {
  production: false,
  firebase: {
    apiKey: "AIzaSyCng_cba2SpK1MVIl478IUInfE5v2UvIxU",
    authDomain: "requisicoes-app.firebaseapp.com",
    databaseURL: "https://requisicoes-app.firebaseio.com",
    projectId: "requisicoes-app",
    storageBucket: "requisicoes-app.appspot.com",
    messagingSenderId: "192989744894"
  }
};
```

Abaixo da propriedade `production` criamos a propriedade `firebase` com as informações copiadas do painel no site do Firebase.

Finalmente, definimos as classes que utilizaremos do `@angularfire`. No arquivo `app.module.ts`, vamos importar os módulos para o banco, autenticação e o arquivo de configuração.

```
import { AngularFireModule } from '@angular/fire';
import { AngularFirestoreModule } from '@angular/fire/firestore';
import { AngularFireAuthModule } from '@angular/fire/auth';
import { environment } from '../environments/environment';
```

Logo em seguida, no array de *imports*, declaramos os módulos `AngularFireModule`, `AngularFirestoreModule` e `AngularFireAuthModule`:

```
imports: [
    BrowserModule,
    AngularFireModule.initializeApp(environment.firebase),
    AngularFirestoreModule,
    AngularFireAuthModule,
  ],
```

Em `AngularFireModule.initializeApp()` passamos a propriedade `environment.firebase` para conexão. Os outros módulos são responsáveis pelo banco e autenticação, respectivamente.

Na seção a seguir implementaremos as classes responsáveis pelo modelo de dados.

3.5 Modelo de dados

Antes de codificar as classes que serão as representações de dados, vamos antever uma questão sobre a manipulação de objetos.

O Firebase só permite a manipulação de objetos JavaScript desserializados (salvar, atualizar), também chamados de objetos literais. Como nossa modelagem definirá objetos customizados, isso significa que, em algum momento, precisamos tratar essa questão.

Para isso, vamos utilizar a biblioteca **class-transform**. No terminal, digite:

```
npm install class-transformer --save
```

Essa biblioteca fornece métodos de serialização e desserialização de objetos.

A primeira classe a ser implementada servirá como base para as demais classes do modelo de dados. Ela terá um método que resolve a conversão para uma classe literal, requisito para salvar dados no Firebase.

Crie as pastas `models` e `core` em `app`. Em seguida, dentro da pasta `core`, criaremos o arquivo **model.ts** com o código a seguir:

```
import { classToPlain } from "class-transformer";

export abstract class Model {
```

```
 id: string;

 toObject(): object {
   let obj: any = classToPlain(this);
   delete obj.id;
   return obj;
 }
}
```

Inicialmente, importamos a biblioteca `class-transformer`, pois no método `toObject()` utilizamos o método `classToPlain(this)` para realizar a desserialização para uma classe literal.

Nossa classe é um modelo, logo fazemos uso da palavra reservada *abstract* e ela possui apenas uma propriedade `id` do tipo `string`.

Vamos criar os modelos, herdando a classe `Model`.

No terminal, após cada linha tecle `enter`:

```
ng g class models/departamento --type=model --skipTests=true
ng g class models/funcionario --type=model --skipTests=true
ng g class models/requisicao --type=model --skipTests=true
```

Serão gerados três arquivos na pasta `models`: `departamento.model.ts`, `funcionario.model.ts` e `requisicao.model.ts`.

O parâmetro `--type=model` é apenas uma convenção para nomenclatura e `--skipTests` está setado para `true` para não criar os arquivos de testes.

Nos arquivos de modelos vamos definir as classes do negócio com suas características.

A primeira classe será escrever a classe que representa o departamento. Abriremos o arquivo `app/src/models/departamento.model.ts` e codificaremos assim:

```
import { Model } from '../core/model';

export class Departamento extends Model {
```

```
 nome: string;
 telefone?: string;
}
```

Depois do nome da classe, `Departamento`, utilizamos a palavra reservada `extends` informando a classe pai *Model*. Dessa forma, herdamos a propriedade e o método definido na classe.

Logo em seguida, declaramos as propriedades específicas do modelo como o nome e o telefone. Note que no atributo `telefone` declaramos assim: `telefone?`, com um sinal de interrogação na frente, indicando que é um campo opcional.

Na classe `Funcionario` (`app/src/models/funcionario.model.ts`) a diferença será somente nos atributos e no *import* da classe `Departamento`.

```
import { Departamento } from './departamento.model';
import { Model } from '../core/model';

export class Funcionario extends Model {
 nome: string;
 funcao: string;
 email: string;
 ultimoAcesso: Date;
 departamento: Departamento;
}
```

Note que a propriedade `departamento` refere-se ao tipo da classe criada anteriormente.

Finalmente, para a classe `Requisicao` (`app/src/models/requisicao.model.ts`), vamos definir duas classes no mesmo arquivo, `Requisicao` e `Movimentacao`. Cada requisição gera um registro de movimento, assim, optei por incluir as definições das classes juntas para mostrar essa possibilidade.

No código a seguir está a implementação da `Requisicao`.

```
import { Departamento } from './departamento.model';
import { Funcionario } from './funcionario.model';
import { Model } from '../core/model';

export class Requisicao extends Model {
 solicitante: Funcionario;
 dataAbertura: any;
 ultimaAtualizacao: any;
 descricao: string;
 status: string;
 destino: Departamento;
 movimentacoes: Movimentacao[];
}
```

Essa classe possui um campo que representa o histórico de movimentações, notado como `movimentacoes: Movimentacao[]`.

Além disso, ela possui as propriedades para a abertura do chamado e última atualização, representados como `dataAbertura: any` e `ultimaAtualizacao: any`, bem como o departamento responsável pela demanda, definido como `destino: Departamento`.

Na classe `Movimentacao`, temos o funcionário que mudou o status da requisição, data e hora e a descrição da movimentação.

```
export class Movimentacao extends Model {
 funcionario: Funcionario;
 dataHora: Date;
 status: string
 descricao: string;
}
```

Pronto, finalizamos a codificação das classes que representam o modelo de negócio do *case* que será implementado.

Podemos observar que o Firebase é uma plataforma completa para o desenvolvimento de aplicações, com integração de autenticação de modo simples a armazenamento de informações.

Além disso, permite que o desenvolvedor não se preocupe com questões de gerenciamento da infraestrutura, já que a solução se encarrega de dimensionar os recursos, conforme aumenta a demanda do uso.

No próximo capítulo, criaremos as classes de serviços para realização de operações no banco e o acesso aos dados.

Capítulo 4
Serviços

Services (serviços) são artefatos que o próprio framework se encarrega de instanciar, além de gerenciar seu ciclo de vida, utilizando o padrão de projeto Singleton.

> **Singleton**
>
> É um padrão de projeto de software (do inglês Design Pattern). Este padrão garante a existência de apenas uma instância de uma classe, mantendo um ponto global de acesso ao seu objeto.

Há uma distinção no Angular entre componentes e serviços a fim de aumentar a modularização e reutilização. O serviço é uma classe com um propósito de separar os papéis quanto à funcionalidade da aplicação.

Nossa primeira codificação será a classe que trata do acesso aos dados. Um dos requisitos apresentados no estudo de caso do sistema é a autenticação do usuário. Somente usuários autenticados podem ler e gravar dados (como vimos no capítulo 3).

O primeiro serviço será responsável justamente pela autenticação do usuário no sistema.

Vamos criá-lo por meio do comando:

```
ng g service services/authentication
```

O comando é composto por:

- `ng g` - gerar o artefato especificado pelo parâmetro seguinte;
- `service` - tipo componente gerado;
- `services/authentication` - diretório/nome da classe para o serviço.

Observe a estrutura gerada:

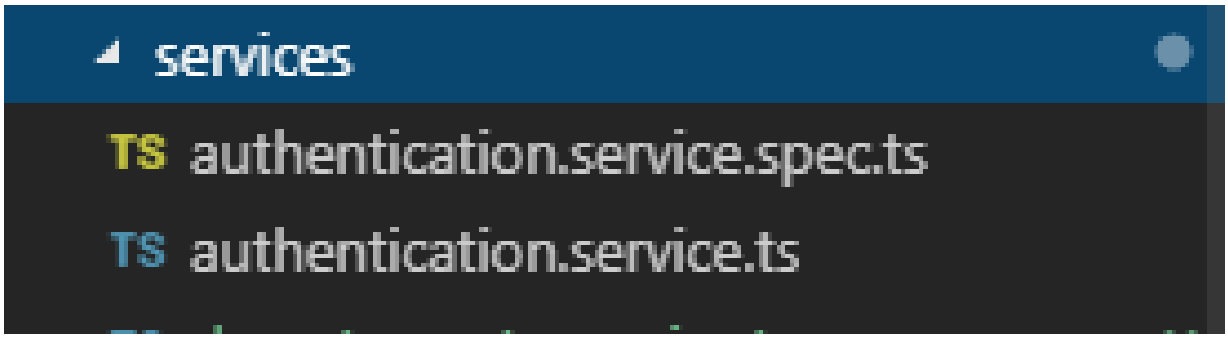


Figura 4.1: Serviços responsável pela autenticação

Uma pasta `services` foi gerada com dois arquivos.

- `authentication.service.spec.ts` - classe para realização de testes.
- `authentication.service.ts` - classe que será responsável pelas regras de autenticação.

Em relação à primeira classe, não faz parte do escopo deste livro o uso de testes unitários (para saber mais, consulte a documentação em https://angular.io/guide/testing). Assim, na criação dos próximos *services* utilizaremos uma flag no comando (`--skipTests`) para não gerar as classes de testes, como fizemos no capítulo anterior.

Vamos escrever a classe `AuthenticationService` a seguir.

4.1 Autenticação no Firebase

Começamos a implementação realizando o `import` da classe `Injectable` . Essa classe permite ao Angular realizar a Injeção de Dependência, controlando a instância assim que outros componentes invocarem suas funções.

Injeção de Dependência

É um padrão de desenvolvimento onde o framework *injeta* em cada componente suas dependências declaradas.

Essa notação utiliza um *decorator* `@` seguido do nome da classe em questão.

Decorator

É tipo especial de declaração que pode ser anexada a uma declaração de classe, método, propriedade ou parâmetro.

```
import { Injectable } from '@angular/core';

@Injectable({
 providedIn: 'root'
})
```

Logo a seguir, a propriedade `providedIn` especifica o nível de injeção, que no caso será na raiz da aplicação (`root`).

O próximo passo são os `imports` da classe `Observable` , `AngularFireAuth` e a lib `firebase` para implementar os métodos de login e logout.

```
import { Observable } from 'rxjs';
import { AngularFireAuth } from '@angular/fire/auth';
import * as firebase from 'firebase/app';
```

Criaremos um objeto `user` do tipo `firebase.User` . Ele será um `Observable` e receberá notificações sempre que ocorrer alguma mudança no estado do usuário. Vamos declará-lo logo abaixo da assinatura da classe.

```
 private user: Observable<firebase.User>;
```

No método construtor, injetamos a classe `AngularFireAuth`. Logo em seguida, passamos o estado de autenticação para o objeto `user`:

```
constructor(private afAuth: AngularFireAuth) {
    this.user = afAuth.authState;
}
```

Com os objetos declarados e a classe `AngularFireAuth` injetada no construtor, nosso próximo passo é implementar os métodos de login e logout do sistema.

4.2 Métodos de login, logout e recuperação de senha

No método de login, passamos dois argumentos do tipo *string* (email e senha) para a classe que injetamos no construtor.

```
login(email: string, password: string):
Promise<firebase.auth.UserCredential> {
    return this.afAuth.signInWithEmailAndPassword(email, password);
}
```

O método `signInWithEmailAndPassword()` recebe os dois argumentos e retorna uma Promise com as credenciais do usuário.

> **PROMISE**
>
> Conceito essencial do JavaScript que representa a conclusão de operações assíncronas.

Já no método `logout()` não passamos nenhum argumento e invocamos a função `signOut()` do `AngularFireAuth` para desconectar.

```
logout(): Promise<void> {
  return this.afAuth.signOut();
}
```

Na próxima função vamos codificar uma opção que permita ao usuário solicitar a recuperação da senha.

```
resetPassword(email: string) {
 return this.afAuth.sendPasswordResetEmail(email)
}
```

Passamos um email no parâmetro para a função `sendPasswordResetEmail()` . Assim o Firebase envia um email com um link para redefinição de senhas para o email informado.

Dessa forma, já temos o primeiro serviço codificado com os métodos responsáveis pela autenticação no sistema.

A seguir, vamos codificar os próximos serviços pensando na modularidade e componentes que poderão ser reutilizáveis.

4.3 Interface genérica de CRUD

Planejar a criação de componentes consistentes e reutilizáveis é um fator importante da construção de sistemas.

Rotinas para criação, leitura, atualização e exclusão de dados, do acrônimo em inglês CRUD (*create*, *read*, *update* e *delete*), são bem comuns, o que pode nos levar a escrever código desnecessariamente.

Outro fator de atenção para iniciantes na programação do framework está no fato de que a maioria dos exemplos na internet sobre CRUD não separa a lógica de operações no Firebase, misturando na classe dos componentes.

Essa prática **não** é recomendada na documentação do framework (https://angular.io/guide/styleguide).

Pensando nisso, vamos separar os papéis de responsabilidade, utilizando também o poder da tipagem e orientação a objetos para escrever interfaces e classes que fornecerão recursos mais flexíveis na construção do sistema.

Na pasta `app/core` clique com o botão direito para criar um arquivo e nomeie-o como `icrud.interface.ts`. A interface `ICrud` é responsável pelos métodos que as classes deverão efetivar para acesso aos dados.

```
import { Observable } from 'rxjs';

export interface ICrud<T> {

 get(id: string): Observable<T>;
 list(): Observable<T[]>;
 createOrUpdate(item: T): Promise<T>;
 delete(id: string): Promise<void>;
}
```

Inicialmente, declaramos a assinatura dos métodos, seus parâmetros, quando existirem, e o tipo de retorno. Na assinatura do método, cotamos como uma interface, e fazemos uso de *Generics*.

> **GENERICS**
>
> O parâmetro de tipo genérico é especificado em colchetes angulares `<>` após o nome da classe. Uma classe genérica pode ter campos genéricos (variáveis de membro) ou métodos.

Para o R do CRUD, há dois métodos, `get()` que recebe um parâmetro do tipo string e um list. O retorno, especificado depois dos dois pontos (`:`) é do tipo Observable, porém o `list()` é um array `<T[ ]>`.

No `createOrUpdate()`, passamos o objeto como argumento `(item:T)` e as funções retornam uma `Promise<T>`. Na implementação do método nós verificamos quando é cada situação, criar ou atualizar.

Finalmente, o método `delete()` recebe um `id` do tipo `string` e seu retorno é do tipo `Promise<void>`.

Finalizada a definição da interface, o próximo passo é criar a classe que implementa os métodos.

4.4 Classe de serviços genérica

Na pasta `core`, crie um arquivo com o nome `iservicefirebase.service.ts`.

Na assinatura da classe vamos declará-la como abstrata, pois servirá como base para outras classes do negócio.

```
export abstract class ServiceFirebase<T extends Model> implements ICrud<T>
{
```

Para isso, identificamos a palavra reservada `abstract`. Além disso, logo depois do nome `ServiceFirebase`, tipamos para uma classe genérica estendendo de umas das classes modelos, conforme apresentada no capítulo 3. Note que também usamos a palavra `implements` seguida de `ICRUD<T>`, o que significa que devemos implementar todos os métodos da interface.

O próximo passo é definir uma referência para a coleção de documentos. Fazemos assim:

```
  ref: AngularFirestoreCollection<T>
```

Utilizamos a classe `AngularFireCollection` da lib `@angular/fire`. Já no método construtor, atribuímos a referência passando o caminho da coleção.

```
  constructor(protected type: { new(): T; }, protected firestore:
AngularFirestore, public path: string) {
    this.ref = this.firestore.collection<T>(this.path);
  }
```

> **@AngularFire**
>
> Para saber mais sobre a implementação, detalhes e exemplos, acesse a documentação em https://github.com/angular/angularfire2/blob/master/docs/firestore/collections.md.

A seguir vamos implementar os métodos. O primeiro método, `get()`, recebe um `id` do tipo `string`. Criamos uma variável do tipo `AngularDocument` para, em seguida, utilizando o operador `map`, da biblioteca Rxjs, retornar um objeto.

```
  get(id: string): Observable<T> {
    let doc = this.ref.doc<T>(id);
    return doc.get().pipe(map(snapshot => this.docToClass(snapshot)));
  }
```

Criamos o método `docToClass()` que utiliza a lib `class-transformer`, serializando o objeto. Um map retorna um `Observable` da classe passada no argumento do tipo.

```
  docToClass(snapshotDoc): T {
    let obj = {
      id: snapshotDoc.id,
      ...(snapshotDoc.data() as T)
    }
    let typed = plainToClass(this.type, obj)
    return typed;
  }
```

No método `list()`, nós utilizamos a referência da coleção (`ref`), invocando o método `valueChanges()`, que retorna um Observable de array da classe genérica.

```
list(): Observable<T[]> {
  return this.ref.valueChanges()
}
```

No trecho abaixo do método `createOrUpdate()` está uma verificação do id para decidir o fluxo para atualização:

```
return this.ref.doc(id).set(obj);
```

ou para inclusão:

```
this.ref.add(obj).then(res => {
      obj.id = res.id;
      this.ref.doc(res.id).set(obj);
    })
```

Observe que estamos passando o id na *arrow function* da Promise em `obj.id = res.id` e depois atualizando objeto. Isso garante que temos um valor criado automaticamente pelo Firebase no campo id do modelo.

Finalmente, o método `delete()` é composto por um argumento id do tipo string.

```
delete(id: string): Promise<void> {
  return this.ref.doc(id).delete();
}
```

O código completo da classe está disponível em https://bit.ly/3aR28Uf.

A partir da referência da coleção, obtemos o documento e invocamos a função `delete` .

Com os modelos, classe e interface do serviço base criado, podemos criar os serviços específicos para cada objeto do nosso modelo com poucas linhas.

No terminal de comando informe:

```
ng g service services/funcionario --skipTests=true
ng g service services/departamento --skipTests=true
ng g service services/requisicao --skipTests=true
```

Criamos três serviços na pasta `services`. Como definimos anteriormente uma classe que já contém a lógica de operação aos dados no Firebase, o que devemos fazer nesse momento é herdar essas funcionalidades.

Assim, a classe `DepartamentoService` (`app/services/departamento.service.ts`) fica:

```
export class DepartamentoService extends ServiceFirebase<Departamento> {

  constructor(firestore: AngularFirestore) {
    super(Departamento, firestore, 'departamentos');
  }
```

Aplicaremos a herança da classe base `ServiceFirebase` usando `extends ServiceFirebase<Departamento>`. No método construtor, chamamos o método `super()`, que recebe a classe modelo `Departamento`, o objeto `firestore` injetada no construtor e o nome que daremos para a coleção.

Uma convenção adotada aqui será nomear a coleção com o nome modelo no plural (`departamentos`).

As outras duas classes de serviços seguem o mesmo padrão. Mudaremos apenas a classe modelo e o nome da coleção, conforme o código a seguir.

Primeiro, para a classe `FuncionarioService` (`app/services/funcionario.service.ts`):

```
export class FuncionarioService extends ServiceFirebase<Funcionario> {

  constructor(firestore: AngularFirestore) {
    super(Funcionario, firestore, 'funcionarios');
  }
}
```

E para a classe do outro serviço `RequisicaoService` (`app/services/departamento.service.ts`):

```
export class RequisicaoService extends ServiceFirebase<Requisicao> {

  constructor(firestore: AngularFirestore) {
    super(Requisicao, firestore, 'requisicoes');
  }
}
```

Assim, nosso projeto já possui as classes de serviços responsáveis pela manutenção dos dados no Firebase. Trataremos no próximo capítulo os componentes que consumirão esses serviços, codificando a lógica e a interface gráfica aos usuários da aplicação.

Capítulo 5
Componentes - Requisito Login

No presente capítulo, codificaremos as interfaces gráficas e toda a lógica para consumir os serviços criados no capítulo anterior.

Desenvolveremos os primeiros requisitos da aplicação, começando pelo login. Para acessar as funcionalidades do sistema, devemos criar um componente onde o usuário possa informar suas credenciais. Apresentaremos uma abordagem na utilização de formulários baseada em modelos.

Nas seções seguintes vamos criar um painel e um menu, trabalhando com rotas, um importante conceito do framework em aplicações SPA.

No quesito de segurança, criaremos guardas que protegerão o acesso às rotas sem a devida autenticação. Além disso, vamos avançar na organização do sistema ao trabalharmos com o compartilhamento de módulos, permitindo flexibilidade, legibilidade e melhor manutenção no código-fonte.

5.1 Login

A primeira implementação será resolver a permissão para acessar as informações, assim vamos gerar o componente `login`.

No final da seção, deveremos ter a seguinte tela funcional:

Figura 5.1: Tela de login

Para criar o componente `login` no terminal digite:

```
ng g component components/public/login --skipTests=true
```

Geração de artefatos e apresentação de códigos

Vou adotar duas práticas a partir desse momento. Para todos os artefatos gerados com Angular CLI a partir daqui, usarei mas *não mencionarei* a flag `--skipTests`. A segunda será durante a explicação do código. Na maioria das vezes, serão omitidos os `imports` das classes dos componentes, evitando um código muito extenso e quebrando a fluidez da leitura. Não se preocupe pois no final de cada seção, ou quando julgar importante, deixarei o link específico do requisito implementado do repositório no GitHub.

Foi gerada uma hierarquia de pastas `components>>public>>login`. A última informação é o nome do componente, no caso, `login`, que contém 3 arquivos:

- **login.component.html**: arquivo HTML onde definimos o template.
- **login.component.css**: arquivo para estilizar os componentes usando CSS.
- **login.component.ts**: classe onde codificamos os métodos, atributos do componente.

Assim que gerada, a classe `AppModule` na pasta `app` foi atualizada com a inclusão de uma declaração do componente login:

```
@NgModule({
  declarations: [
    AppComponent,
    LoginComponent,
  ],
```

Uma das vantagens de utilizar o Angular CLI é a declaração automática dos componentes no módulo principal.

Primeiro, vamos utilizar uma biblioteca que fornece um *popup* customizável, responsível e muito bonito na exibição de alertas e mensagens, o *SweetAlert2*.

No terminal de comandos, informe:

```
npm install --save sweetalert2
```

Para utilizá-lo na classe, basta declarar o `import` da biblioteca:

```
import Swal from 'sweetalert2'
```

Vamos iniciar o código pela classe do componente login, definindo as propriedades e métodos do login, porém, antes, analisaremos o código gerado da classe `LoginComponent` (`app/components/public/login/**login.component.ts`).

Abaixo do `import` temos o seguinte código:

```
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
```

A diretiva `@Component` identifica a classe imediatamente abaixo como uma classe de componente e especifica seus metadados:

- **selector**: seletor CSS que diz ao Angular para criar e inserir uma instância desse componente onde quer que encontre a tag correspondente no modelo HTML. No caso `<app-login> </app-login>` , o framework insere uma instância do componente entre as tags.
- **templateUrl**: endereço relativo do template HTML.
- **styleUrls**: endereço relativo do arquivo de estilos do componente.

Começando a codificação da classe:

```
export class LoginComponent implements OnInit {

  email: string;
  senha: string;
  mensagem: string;
  emailEnviado: boolean;
```

Logo abaixo da assinatura `export class LoginComponent` , vamos declarar três atributos. Todos são do tipo string: `email: string` , `senha: string` , `msg: string` . E uma propriedade para controlar a exibição da mensagem quando solicitarmos a recuperação do e-mail: `emailEnviado: boolean` .

No método construtor precisamos injetar o serviço responsável pelo acesso, `AuthenticationService` , e utilizar a classe `Router` para controlar a navegação. Escrevemos o método assim:

```
  constructor(private authServ: AuthenticationService, private router:
Router) { }
```

O próximo passo é implementar o método para `logar()` . Primeiro, verificaremos o preenchimento dos atributos `email` e `senha` , que representam os campos de entrada.

Atribuímos uma mensagem caso um dos campos não esteja preenchido.

```
logar() {
    try {
      if (this.email == undefined ||
        this.senha == undefined) {
        this.mensagem = 'Usuário ou senha vazios'
        return
      }
```

Na sequência, com o objeto `authServ` , invocamos o método do serviço informando no parâmetro o `email` e a `senha` :

```
      this.authServ.login(this.email, this.senha)
        .then(() => {
```

```
        this.router.navigate(['/admin/painel'])
      })
```

Depois do código `.then(() >=` , escrevemos na Promise, utilizando o objeto `router` para navegar até a rota `/admin/painel` . Essa rota será definida posteriormente na classe responsável pelo mapeamento das rotas.

Agora, tratamos as exceções com o `catch(erro =>)` assim:

```
.catch(erro => {
        let detalhes = '';
        switch (erro.code) {
          case 'auth/user-not-found': {
            detalhes = 'Não existe usuário para o email informado';
            break;
          }
          case 'auth/invalid-email': {
            detalhes = 'Email inválido';
            break;
          }
          case 'auth/wrong-password': {
            detalhes = 'Senha Inválida';
            break;
          }
          default: {
            detalhes = erro.message;
            break;
          }
        }
          this.mensagem = `Erro ao logar. ${detalhes}`;
      });
```

Com o operador de controle `switch` , nós verificamos os tipos de erros que podem surgir ao logar no Firebase, atribuindo uma mensagem mais amigável para a variável `detalhes` .

Finalizamos a implementação tratando o `catch` genérico do `try` logo no início da implementação:

```
} catch (erro) {
      this.mensagem = `Erro ao logar. Detalhes: ${erro}`;
    }
```

Finalmente, escrevemos o método `enviaLink()`. Essa função permite o processo de recuperação de senhas ao deixar que a plataforma se encarregue de enviar e-mails para inicialização de uma nova senha.

```
 async enviaLink() {
    const { value: email } = await Swal.fire({
      title: 'Informe o email cadastrado',
      input: 'email',
      inputPlaceholder: 'email'
    })
```

Antes do nome do método, incluímos a palavra reservada `async`, que define uma função como assíncrona. Para acessarmos os recursos da biblioteca `Sweetalert2`, fazemos o `import`, como visto na instalação. A partir disso podemos utilizar os métodos através do objeto `Swal`.

Definimos algumas propriedades para a exibição do *popup* como `title`, `input` e `inputPlaceholder`.

O método `fire` é responsável por lançar os alerts com base nas configurações das propriedades informadas na sequência.

Logo em seguida, verificamos se o campo `email` foi preenchido, invocando o método do serviço que solicita o envio da senha:

```
if (email) {
      this.authServ.resetPassword(email)
        .then(() => {
          this.emailEnviado = true;
            this.mensagem = `Email enviado para ${email} com instruções
para recuperação.`
        })
        .catch(erro => {
          this.mensagem = `Erro ao localizar o email. Detahes
${erro.message}`
```

```
        })
    }
  }
```

O atributo definido como `this.emailEnviado = true;` será responsável pela exibição da mensagem atribuída na sequência. Tratamos também o erro com `catch()`, exibindo os detalhes do erro.

O resultado dessa implementação do comando será:

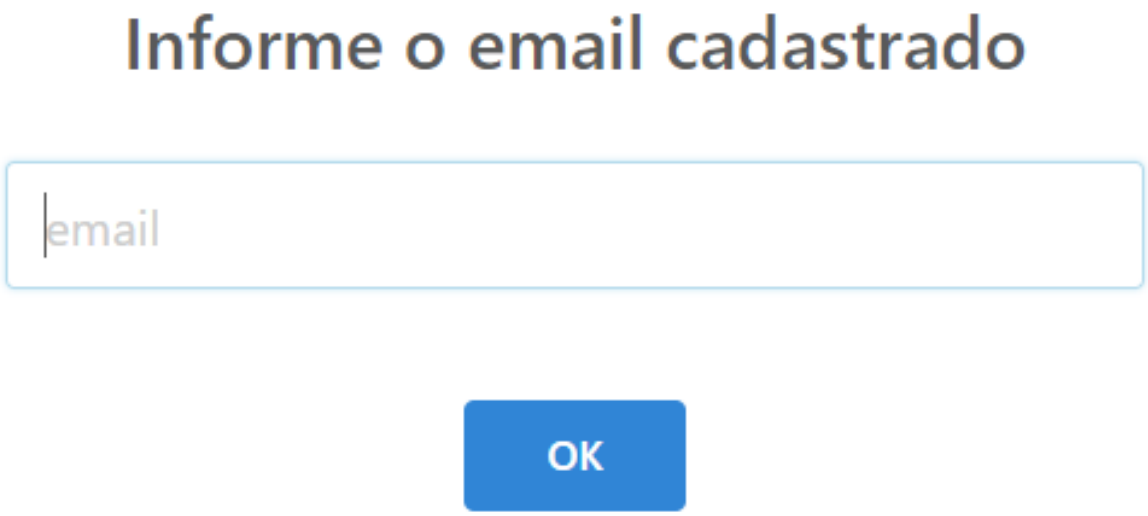


Figura 5.2: Popup para recuperar senha

O próximo passo é obrigatório para manipular formulários. Devemos importar no módulo principal a classe `FormsModule` na seção `import` em `app/app.módulo.ts`:

```
import { FormsModule } from '@angular/forms';
```

Depois fazemos a declaração da classe no array de `imports` no mesmo arquivo:

```
imports: [
    //Outros imports omitidos
     FormsModule,
```

Já temos a lógica do componente implementada. Vamos codificar o template `html` da classe para exibir o formulário do login com os campos de entradas e botões de ação.

5.2 Template Driven - Formulário de Login

Há duas abordagens no Angular para trabalhar com formulários.

Os formulários reativos (**Reactive Forms**) são caracterizados por serem reutilizáveis e escalonáveis e seu uso está relacionado a construções de aplicações que já utilizam o padrão reativo.

Já os formulários orientados por modelos (**Template Driven**) são indicados para adicionar um formulário simples a um aplicativo, como um formulário de login. Será exatamente a nossa implementação a seguir.

Abra o arquivo do template no caminho
`app/components/public/loginlogin.component.html` .

O código completo do template está disponível em: https://bit.ly/2PGJIM2, assim focaremos nas particularidades do framework, já que detalhes do HTML não estão no escopo do livro.

A seguir, o código do campo para caixa de entrada do email:

```
<input class="form-control" name="emailControle" #emailControle="ngModel"
type="text" placeholder="Informe o Email"
   [(ngModel)]="email"  maxlength="60" email required>
```

Criamos uma variável `#emailElement` e referenciamos ao `ngModel` . Usamos também a notação **two-way data binding**, representada com `[(ngModel)]="variável"` , informando o email na variável.

Isso garante que a atualização ocorra no template e na classe do componente a qualquer momento, independente do lugar onde a

alteração se iniciou.

Na sequência, vamos exibir as mensagens de validação do campo `email`:

```
<div  *ngIf="emailControle.invalid" class="text-danger">
    <div  *ngIf="emailControle.errors.required">Email é obrigatório</div>
    <div  *ngIf="emailControle.errors.email">Email inválido</div>
</div>
```

Conseguimos acompanhar as alterações de estado e a validade dos controles de formulário. Usamos a diretiva de controle `*ngIf` para verificar a validade da variável `emailcontrole.invalid`. Se verdadeiro, verificamos a obrigação do preenchimento com `emailControle.errors.required` e se é um campo do tipo `email` com `emailControle.errors.email`.

O resultado da validade é apresentado conforme o usuário interage com o formulário:

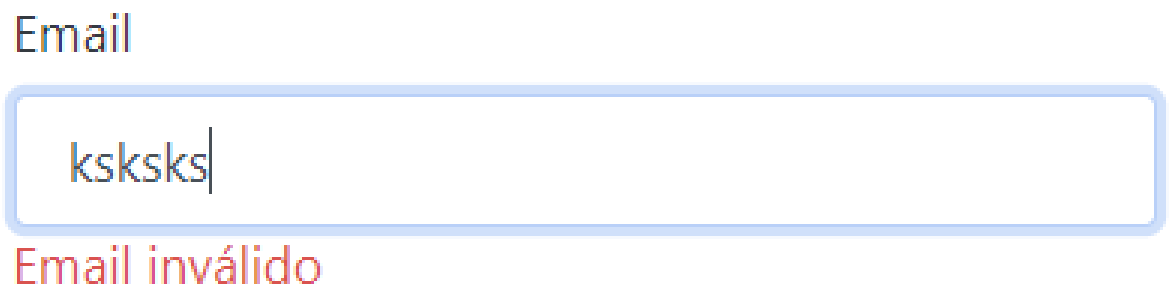


Figura 5.3: Validação para email

Fornecemos, ainda, feedback visual usando classes CSS no primeiro `div` em `class="text-danger"`, definindo o texto na cor vermelha.

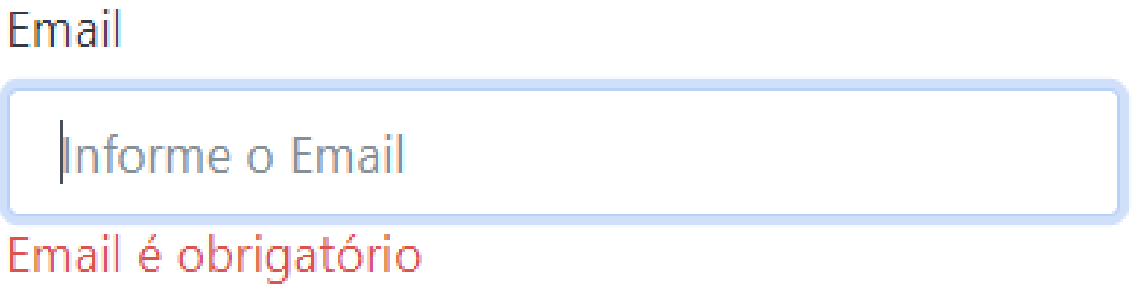


Figura 5.4: Campo obrigatório

O campo da senha segue a mesma lógica. Definimos uma variável de template `#passwordControl` e as propriedades de um campo de senha comum como `type="password"` e o *two-way data binding* para o atributo `senha` da classe `login`:

```
 <input name="passwordControl"  #passwordControl="ngModel"
[(ngModel)]="senha" type="password" class="form-control"
placeholder="Senha" required minlength="6" maxlength="30">
```

Na sequência, vamos codificar os botões. O primeiro botão será responsável pela submissão do formulário para logar.

```
 <button type="submit" class="btn btn-info btn-block" (click)="logar()">
 <i class="fas fa-sign-in-alt"></i> Logar</button>
```

Usamos a expressão `( )` para indicar que é um *event binding* para o evento `click` do botão. Em seguida, atribuímos ao método `logar()` já implementado na classe do componente.

EVENT BINDING

Utilizado para passar os dados do template para a classe do componente. Como exemplos de eventos temos pressionamentos de tecla, movimentos do mouse, cliques e toques.

E, finalmente, codificamos o botão para enviar o link de recuperação de senha, informando o método `enviaLink()` :

```
<button type="submit" (click)="enviaLink()" class="btn btn-lg btn-success
btn-block text-uppercase"> <i class="far fa-envelope"></i>
Recuperar</button>
```

Para visualizar o código de estilização do componente login acesse: https://bit.ly/2DLnAev.

Finalizamos a construção do componente login. Falta definir uma rota para aplicação iniciar o projeto na página de login.

Encontre o arquivo `app-routing.module.ts` na pasta `app` . Na classe `AppRoutingModule` definimos as rotas da aplicação. No array de rotas `routes: Routes` , vamos definir dois `path` :

```
routes: Routes = [
  { path: '', redirectTo: 'login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
```

O primeiro `path` é um *alias* para indicar quando o usuário navega na raiz da url, sem passar nenhuma informação. É o endereço que o aparece quando executamos a aplicação com o comando `ng serve -o` .

No segundo caminho nomeamos a rota como `'login'` e vinculamos ao componente `LoginComponent` .

Nesse momento já temos o login funcional. Porém, ao executar o projeto (`ng serve --o`) e inserir o e-mail e a senha criados no capítulo do Firebase (cap. 3), o console da página mostrará um erro, já que redirecionamos para uma rota na classe do componente:

```
    this.router.navigate(['/admin/painel'])
```

O caminho `/admin/painel` ainda não existe!

Agora podemos criar o menu que ficará disponível em toda aplicação.

5.3 Menu da aplicação

No final da seção teremos o seguinte componente:

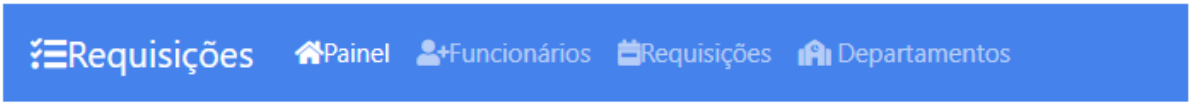


Figura 5.5: Menu da aplicação

Para isso, vamos começar gerando o componente no terminal utilizando o comando:

```
ng g c components/admin/menu
```

Para que ele apareça em toda a aplicação, devemos incluir o seletor do componente `<app-menu></app-menu>` no componente principal em `app.component.html` .

```
<div class="col-lg-12 py-3">
  <app-menu></app-menu>
</div>
<div class="bg-color">
  <router-outlet></router-outlet>
</div>
```

Esse nome está no `selector` da classe do componente (`components/admin/menu/menu.component.ts`).

A diretiva `router-outlet` marca o local em que o roteador exibe os componentes dinamicamente.

Abrindo a classe do componente, vamos declarar um objeto `user` e injetar duas classes no construtor:

```
  user: Observable<firebase.User>;
  constructor(private authServ: AuthenticationService, private router:
Router) { }
```

Com o objeto, monitoramos o estado da autenticação e no método construtor, criamos dois objetos que serão utilizados para implementar os métodos a seguir.

Começaremos com `ngOnInit()`, que é um dos métodos do ciclo de vida do framework que inicializa o componente após o Angular exibir primeiro as propriedades vinculadas a dados e definir as propriedades de entrada da diretiva ou componentes.

Vamos utilizar com frequência esse método para inicializar os objetos e invocações de métodos.

Nosso objetivo é monitorar o estado da autenticação:

```
ngOnInit() {
    this.user = this.authServ.authUser();
  }
```

Dessa forma, conseguimos monitorar o objeto `user`, pois a função `authUser()` retorna um `Observable` do usuário no Firebase.

Agora, podemos codificar uma função para encerrar a sessão. Fazemos isso com o código:

```
sair() {
    this.authServ.logout().then(() => this.router.navigate(['/']));
  }
```

Invocamos o método `logout()` do serviço injetado no construtor e na realização da *Promise* `then() =>`, navegamos até a rota raiz `navigate(['/'])`, redirecionando para o login.

Desenvolvendo agora o código do template, disponível em https://bit.ly/2J76awB, vale destacar o seguinte trecho do código:

```
<li class="nav-item" routerLinkActive="active">
       <a class="nav-link" routerLink="/" href="#">
         <i class="fas fa-home"></i>Painel</a>
     </li>
```

`routerLink` é o local onde especificamos as rotas da aplicação, sempre iniciando com `/`. Em `routerLinkActive` definimos uma classe CSS para quando o link da rota se tornar ativo.

Para o botão `sair`, fazemos um *event binding* em:

```
<a href="#" (click)="sair()" class="nav-link">SAIR</a>
```

Ao servir a aplicação (`ng serve --o`), você tem o seguinte resultado:

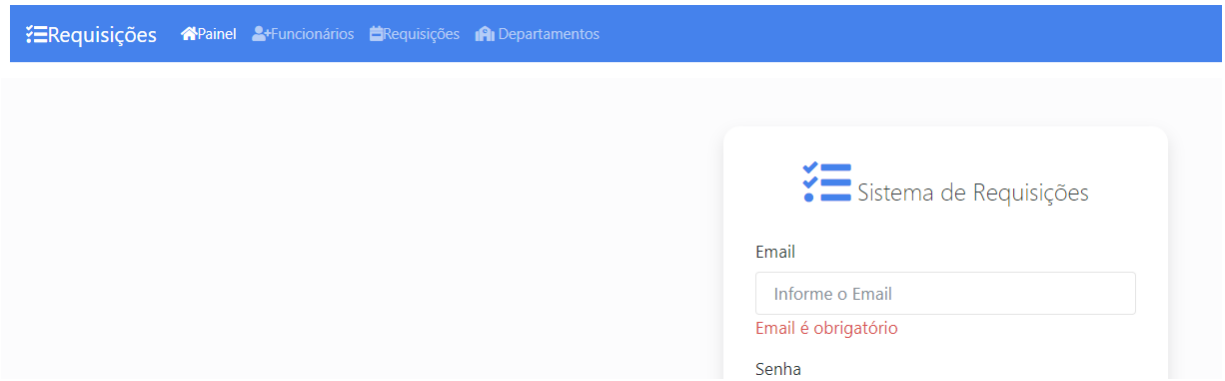


Figura 5.6: Login, aparecendo o menu

Observe que a tela do login já exibe o menu com links para outros componentes, o que não faz muito sentido! Vamos controlar a exibição do componente utilizando a diretiva `*ngIf`.

No início do template:

```
<nav *ngIf="(user|async)?.uid"
```

Verificamos dentro da tag que renderiza o menu de navegação se existe a propriedade `uid` do objeto `user`. Como é do tipo Observable, devemos usar o *pipe* (o sinal de barra de pé `|`) e a palavra `async`.

PIPE

Um pipe recebe dados como entrada e os transforma em uma saída desejada. Podemos criar um pipe personalizado.

Nas próximas implementações de componentes mostrarei uma abordagem na construção com vistas a projetos de médio e grande portes.

5.4 Painel administrativo - Componentes com Lazy Loading

Os primeiros componentes que implementamos, login e menu, foram adicionados ao módulo principal. Conforme nossa aplicação aumenta, essa forma não é uma prática recomendada, já que todos os componentes são carregados na inicialização.

Assim, vamos trabalhar com o conceito de **Lazy Loading**, dividindo nossa aplicação em módulos, que serão carregados somente quando houver necessidade, isto é, quando o usuário navegar até a rota daquele módulo.

Entre outras vantagens podemos citar o impacto do tempo de carregamento do projeto.

Para utilizar a técnica devemos realizar os seguintes procedimentos:

1. Criar o módulo usando `ng g module nome-do-modulo`;
2. Criar o componente `ng g c nome-do-componente`;
3. Configurar a rota.

Faremos isso para o componente painel. No terminal, informe o comando:

```
ng g m components/admin/painel --routing
```

Utilizamos o Angular CLI com `ng g` para gerar um módulo com o comando abreviado `m` nas pastas `components/admin` e nomeamos o módulo como `painel`.

A flag `--routing` cria um arquivo de rotas do componente chamado `painel-routing.module.ts` .

Agora, criaremos o componente. No terminal:

```
ng g c components/admin/painel
```

Trocamos o `m` por `c` , gerando um componente.

A estrutura da pasta `painel` contém 5 arquivos:

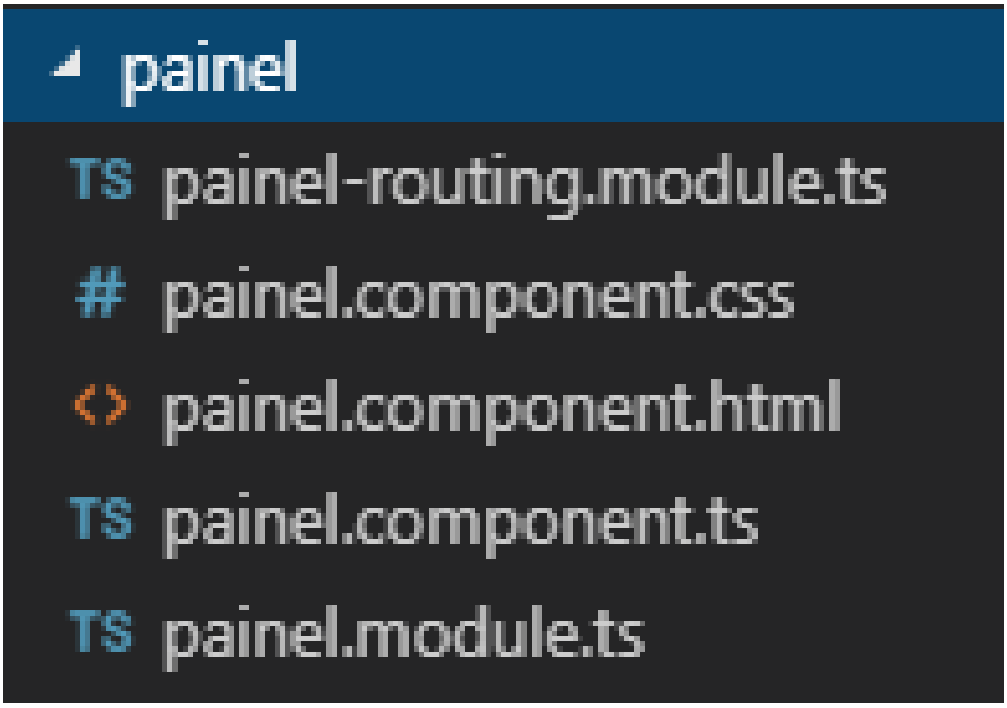


Figura 5.7: Estrutura da pasta painel

Agora, podemos abrir arquivo principal de rotas, `app-routing.module.ts` , na pasta `app` e incluir uma rota no array.

```
  { path: 'admin/painel', loadChildren: () =>
import('./components/admin/painel/painel.module')
  .then(m => m.PainelModule), canActivate: [AuthguardService]},
```

Definimos o `path` como `admin/painel` , exatamente como fizemos na implementação do método login. A diferença agora é que não

associamos a rota ao componente, mas ao módulo, utilizando a propriedade `loadChildren` em `.then(m => m.PainelModule)`.

Note que a sintaxe usa `loadChildren` seguida por uma função que usa a sintaxe de importação integrada ('...') do navegador para *importações dinâmicas*. O caminho de importação é o caminho relativo para o módulo.

Vale destacar que esse recurso é uma nova forma de configurar rotas utilizando lazy, introduzida a partir na versão 8.

> **IMPORTAÇÕES DINÂMICAS**
>
> A importação dinâmica é útil em situações em que você deseja carregar um módulo condicionalmente ou sob demanda. Além disso, essa forma melhora o suporte de editores que podem entender e validar as importações.

Desta forma, o roteador sabe ir ao módulo do recurso. Os módulos é que ficarão responsáveis pelo carregamento dos componentes.

5.5 Protegendo as rotas com Guardas

O objetivo desta seção é apresentar uma estratégia para proteger as rotas.

Da forma como implementamos até aqui, o usuário consegue acessar qualquer página (componente) informando diretamente a rota na url, sem estar logado.

Vamos criar um serviço que verifica se o funcionário está autenticado. No terminal, informe:

```
ng g s services/authguard
```

Abrindo o arquivo `authguard.service.ts` da pasta `services`, vamos implementar a interface `CanActivate`, importando de `@angular/router`.

Essa interface auxilia no gerenciamento da navegação, decidindo se uma rota pode ser ativada.

```
import { CanActivate} from '@angular/router';

export class AuthguardService implements CanActivate{
```

Realizamos a injeção de duas classes para redirecionar a navegação (`Router`) e observamos o estado da autenticação com `AngularFireAuth` :

```
constructor(private afAuth: AngularFireAuth, private router: Router) { }
```

Vamos implementar o método `canActivate`, retornando `false` se não existir um usuário, e redirecionando a navegação para a rota `/login`.

```
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.afAuth.authState.pipe(
      take(1),
      map(user => !!user),
      tap(loggedIn => {
        if (!loggedIn) {
          this.router.navigate(['/login']);
        }
      })

    )
  }
```

Para finalizar, devemos registrar no arquivo de rotas quais delas serão protegidas. Editamos o arquivo `app-routing.module.ts`, informando a propriedade `canActivate`, dessa maneira:

```
 { path: 'login', component: LoginComponent },
  { path: 'admin/painel', loadChildren: () =>
```

```
import('./components/admin/painel/painel.module')
  .then(m => m.PainelModule), canActivate: [AuthguardService]},
```

No path inicial alteramos para `admin/painel` e na frente do módulo do painel informamos `canActivate: [AuthguardService]`, passando nossa implementação da guarda.

Assim, ao subir aplicação, se o usuário já estiver logado, a url raiz será a do componente painel. Caso contrário, o método retorna `false`, redirecionando para a rota do componente login.

Conforme aumenta a complexidade da aplicação, uma prática recomendada é criar módulos compartilhados.

Portanto, o próximo passo é arranjar os artefatos que utilizaremos com certa frequência, como os componentes da biblioteca PrimeNG e outras classes em módulos, visando a organização do projeto e agilidade no código.

5.6 Organizando e compartilhando módulos

Vamos criar um módulo que vai expor os componentes do PrimeNG.

No terminal, informe o comando:

```
ng g m modules/primeNG
```

Na pasta `modules` edite o arquivo `primeng.module.ts`.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { ButtonModule } from 'primeng/components/button/button';
import { FieldsetModule } from 'primeng/components/fieldset/fieldset';

// Alguns imports
@NgModule({
  imports: [
    CommonModule,ButtonModule,FieldsetModule,
```

```
InputMaskModule,MessagesModule, CheckboxModule ,

DataTableModule,DialogModule,InputTextModule,InputTextareaModule,DropdownM
odule,
    ConfirmDialogModule,CalendarModule,TabViewModule, ToggleButtonModule
  ],
  exports:[
    ButtonModule,FieldsetModule, InputMaskModule,MessagesModule,
CheckboxModule ,

DataTableModule,DialogModule,InputTextModule,InputTextareaModule,DropdownM
odule,
    ConfirmDialogModule,CalendarModule,TabViewModule, ToggleButtonModule


  ],
  declarations: []
})
export class PrimeNGModule { }
```

Realizamos os `imports` dos componentes que utilizaremos no decorrer do desenvolvimento dos componentes.

Logo em seguida, registramos nos arrays de `imports` e `exports` os módulos dos componentes.

Agora, podemos gerar também mais um módulo mais genérico para centralizar os `imports` dos componentes:

```
ng g m modules/comum
```

Ainda na pasta `modules`, agora temos o arquivo `comum.module.ts`. Vamos incluir os seguintes módulos no array de `imports` e `exports`:

```
@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    PrimeNGModule
  ],
  exports:[
    CommonModule,
```

```
    FormsModule,
    PrimeNGModule
  ],
```

Declaramos o `CommonModule` que fornece as diretivas básicas do framework como `*ngIf` e `*ngFor`, o módulo para trabalhar com formulários `FormsModule` e o módulo que criamos anteriormente `PrimeNGModule`.

Link para o código completo do módulo: https://bit.ly/2WbYxvS.

Já temos o essencial para construir componentes mais complexos, com reutilização de código e mais organização do projeto.

Continuando o desenvolvimento, no próximo capítulo vamos implementar o componente responsável pela gerência de departamentos.

Capítulo 6
Formulários reativos e Pipe - Departamento e Funcionário

Neste capítulo vamos trabalhar com formulários reativos, pontuando as vantagens dessa abordagem e ganhos em relação à perspectiva anterior.

Utilizaremos essa abordagem para implementar o requisito *Cadastrar Departamento*.

Finalmente, finalizaremos o capítulo com as seções sobre o componente para o Funcionário. Implementaremos um tipo especial de classe, pipe, que nos auxiliará na visualização filtrada dos registros.

6.1 Componente Departamento

Para o próximo componente, vamos gerar o módulo e depois o componente, utilizando Lazy Loading.

No terminal, informamos os comandos (depois de cada linha pressione `enter`):

```
ng g m components/admin/departamento --routing
ng g c components/admin/departamento
```

Seguimos incluindo uma rota para o departamento em `app-routing.module.ts`:

```
  { path: 'admin/departamento', loadChildren: () =>
import('./components/admin/departamento/departamento.module')
  .then(m => m.DepartamentoModule), canActivate: [AuthguardService]},
```

Observe que declaramos a guarda na propriedade `canActivate: [AuthguardService]`.

Escrevemos a rota no template do componente menu (`login.component.html`):

```
<a class="nav-link" routerLink="/admin/departamento" href="#">
```

E no módulo do departamento (`departamento-routing.module.ts`) associamos o path ao componente.

```
const routes: Routes = [
 { path: '', component: DepartamentoComponent }
];
```

Com a geração e a configuração do componente, podemos iniciar a codificação da classe.

No final da seção, teremos o componente com o seguinte visual:

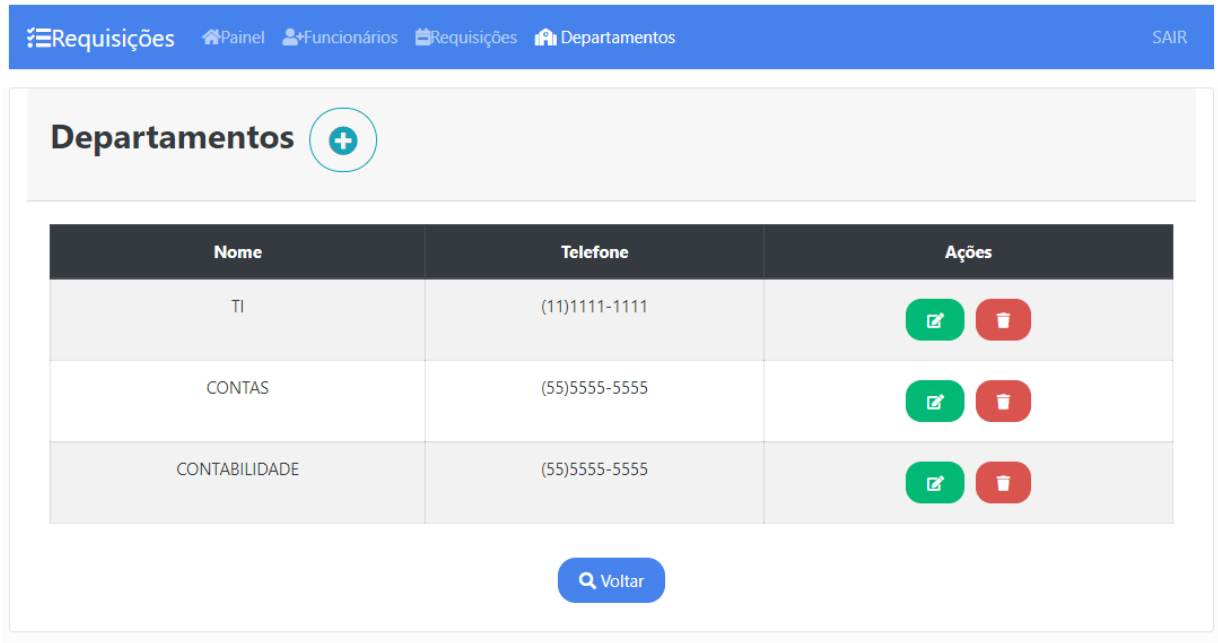


Figura 6.1: Componente Departamento

Essa interface será o padrão de layout adotado durante todo o projeto para atender os requisitos do *case* proposto.

Neste capítulo, veremos outra abordagem para formulários, conhecida como formulários reativos (*Reactive Forms*).

Formulários reativos

Formulários reativos são uma boa opção, pois apesar da lógica de validação mais complexa é realmente mais simples de implementar, com o ganho de podermos testar a lógica de validação de formulário.

> ### Reactives Forms vs. Template Driven
>
> As duas abordagens possuem vantagens, mas em geral, é melhor escolher uma das duas formas de fazer formulários e usá-las consistentemente em todo o aplicativo.

O primeiro passo é importar e registrar o módulo `ReactiveFormsModule` em `departamento.module.ts` .

```
@NgModule({
 declarations: [DepartamentoComponent],
 imports: [
   SharedModule,
   ReactiveFormsModule,
   DepartamentoRoutingModule
 ]
```

Realizamos também a declaração de `SharedModule` . Esse, por sua vez, exporta outros módulos necessários ao componente, como os módulos de cada componente visual do **PrimeNG** que utilizaremos no decorrer do desenvolvimento.

Na classe do componente (`departamento.component.ts`), declaramos 4 atributos para criar a tabela, formulário e ações.

```
 departamentos$: Observable<Departamento[]>;
 edit: boolean;
```

```
displayDialogDepartamento: boolean;
form: FormGroup;
```

Em `departamentos$`, temos um objeto do tipo `Observable<Departamento[]>` que usaremos para apresentar os registros salvos no Firebase. O uso de `$` no nome é uma convenção para identificar objetos que utilizam operações assíncronas.

Na variável `edit`, controlamos o modo de inclusão ou edição para apresentar a legenda correta no botão. Já a propriedade `displayDialogDepartamento`, também booleana, representa a visibilidade de um componente *dialog* que codificaremos no template.

Por fim, a variável `form` representa a instância de componentes do formulário.

Seguindo o código:

```
constructor(private departamentoService: DepartamentoService, private fb: FormBuilder) { }
```

Temos a injeção do serviço de departamento `departamentoService` e a classe `FormBuilder`, que nos permite criar formulários complexos sem a necessidade de vários controles.

Para definir os campos do formulário e configurar as regras de validação, escrevemos o método `configForm()`:

```
configForm() {
   this.form = this.fb.group({
     id: new FormControl(),
     nome: new FormControl('', Validators.required),
     telefone: new FormControl('')
   })
 }
```

Através do método `group`, criamos uma instância do formulário, passando uma coleção de `FormControl` para cada atributo. No

construtor do atributo `nome`, passamos uma validação, informando o preenchimento obrigatório do campo em `Validators.required`.

No endereço https://angular.io/api/forms/Validators, encontramos uma relação dos tipos de regras já definidas como máximo, mínimo, ou ainda, podemos definir nossas próprias regras.

Na sequência, declaramos o método na inicialização, em `ngOnInit()`:

```
  ngOnInit() {
   this.departamentos$ = this.departamentoService.list()
   this.configForm()
 }
```

Preenchemos o campo `departamentos$`, invocando o método `list()` do serviço.

A seguir, vamos criar o método que aciona o *dialog* para incluirmos um departamento:

```
add() {
   this.form.reset()
   this.edit = false;
   this.displayDialogDepartamento = true;
 }
```

Com `form.reset()` reiniciamos o estado do formulário de todos os campos para nulo.

Setamos o atributo `edit` para `false`, indicando que não estamos em edição, e `true` para `displayDialogDepartamento`, exibindo o componente que possuirá essa propriedade.

O próximo método é responsável pela seleção do objeto na tabela.

```
 selecionaDepartamento(depto: Departamento) {
   this.edit = true;
   this.displayDialogDepartamento = true;
   this.form.setValue(depto)
 }
```

Passamos no parâmetro da função o objeto selecionado, `depto: Departamento`, a partir do template. Mudamos para o modo de edição com `this.edit = true` e exibimos o *dialog* com `this.displayDialogDepartamento = true;`.

Finalmente, passamos o objeto para o formulário através do método `setValue(depto)`.

Para persistir os dados do formulário, implementaremos o método `save()`:

```
save() {
this.departamentoService.createOrUpdate(this.form.value)
.then(() => {
  this.displayDialogDepartamento = false;
  Swal.fire(`Departamento ${!this.edit ? 'salvo' : 'atualizado'} com
sucesso.`, '', 'success')})
.catch((erro) => {
 this.displayDialogDepartamento = false;
 Swal.fire(`Erro ao ${!this.edit ? 'salvo' : 'atualizado'} o
departamento.`, `Detalhes: ${erro}`, 'error') })
  this.form.reset()
 }
```

Explicando o código, passamos o formulário para o método do serviço que cria ou atualiza os dados do formulário: `createOrUpdate(this.form.value)`, lembrando que a regra para descobrir quando é cada ação, inserção ou atualização, baseia-se na existência de um valor para o id, explicada no capítulo de serviços.

No `then(() =>)` da Promise, escondemos o *dialog* com `this.displayDialogDepartamento = false` e exibimos uma mensagem de *alert* com a biblioteca *SweetAlert2* no método `Swal.fire`. Esse método recebe um título, uma mensagem e um tipo `success`.

Aqui vale a menção do uso de **template string** `${váriavel}` para concatenar a mensagem, além de usarmos o operador ternário `?`

para exibir a mensagem verificando o modo do formulário, dessa forma:

```
${!this.edit ? 'salvo' : 'atualizado'}
```

Em caso de erro, capturamos com `catch` e exibimos como detalhes usando `${erro}`, e o tipo do *dialog* para `error`.

Por fim, escrevemos o código para excluir um departamento:

```
delete(depto: Departamento) {
  Swal.fire({
    title: 'Confirma a exclusão do departamento?',
    text: "",
    type: 'warning',
    showCancelButton: true,
    confirmButtonText: 'Sim',
    cancelButtonText: 'Não'
  }).then((result) => {
    if (result.value) {
      this.departamentoService.delete(depto.id)
        .then(() => {
          Swal.fire('Departamento excluído com sucesso!', '', 'success')
        })
    }
  })
}
```

No começo do método temos a configuração das propriedades para exibir uma mensagem de confirmação, antes de efetivar a exclusão.

Informamos os rótulos para o botão de confirmação, `confirmButtonText: 'Sim'`, e cancelamento, `cancelButtonText: 'Não'`.

Em seguida, verificamos na Promise o resultado do *click* `if (result.value)` e invocamos o método do serviço, passando o id do departamento, `departamentoService.delete(depto.id)`. Concluímos com a mensagem `'Departamento excluído com sucesso!`.

E assim, temos a lógica do componente departamento implementada. O código completo está disponível em

https://bit.ly/2VFd1oo.

Vamos codificar o template no html correspondente.

6.2 Template do Departamento - Recuperando e exibindo informações

Vamos começar o código em `departamento.component.html` definindo um cabeçalho com um botão para acionar um novo registro:

```
<div class="card-header">
     <h3> Departamentos
       <button type="button" style="margin-right: 0px" (click)="add()"
class="text-right btn btn-outline-info btn-lg">
         <i class="fa fa-plus-circle" aria-hidden="true"></i>
       </button>
     </h3>
   </div>
```

Usamos a classe `card-header` do **Bootstrap** para estilizar o cabeçalho. E associamos o *click* botão ao método `add()` desenvolvido na classe do componente.

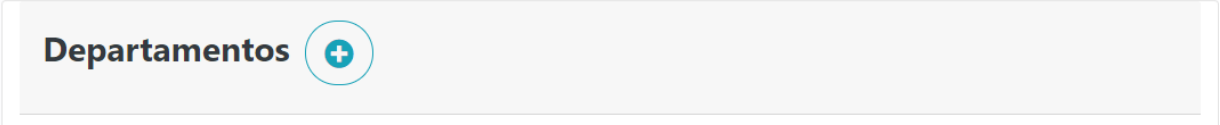


Figura 6.2: Cabeçalho Departamento

A seguir, codificamos a tabela:

```
 <table class="table table-striped table-hover table-bordered col-
centered">
       <thead class="thead-dark">
         <tr>
           <th class="text-center">Nome</th>
           <th class="text-center">Telefone</th>
```

```
            <th class="text-center">Ações</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let departamento of departamentos$ | async">
            <td class="text-center">{{departamento.nome}}</td>
            <td class="text-center">{{departamento.telefone}}</td>
            <td class="text-center">
              <button type="button"
(click)="selecionaDepartamento(departamento)" class="btn btn-success  ">
                <i class="fas fa-edit"></i>
              </button>
              <button type="button" (click)="delete(departamento)"
class="btn btn-danger  ">
                <i class="fas fa-trash"></i>
              </button>
            </td>
          </tr>
        </tbody>
      </table>
```

Definimos as colunas `Nome`, `Telefone` e `Ações` e, na tag `<tr>`, utilizamos a diretiva `*ngFor` para criar cada linha da tabela. Nomeamos cada objeto como `departamento` e usamos o operador `| async` para iterar sobre `departamento$`.

Na terceira célula, criamos dois botões. O primeiro é responsável pela edição. Vinculamos ao método `selecionaDepartamento(departamento)` e mostramos um ícone da biblioteca *FontAwesome*, `fas fa-edit`.

O segundo botão, vinculamos à função de excluir `delete(departamento)`, também implementada na classe, passando a variável `departamento` no parâmetro.

A imagem a seguir apresenta o resultado do *click* no botão de exclusão:

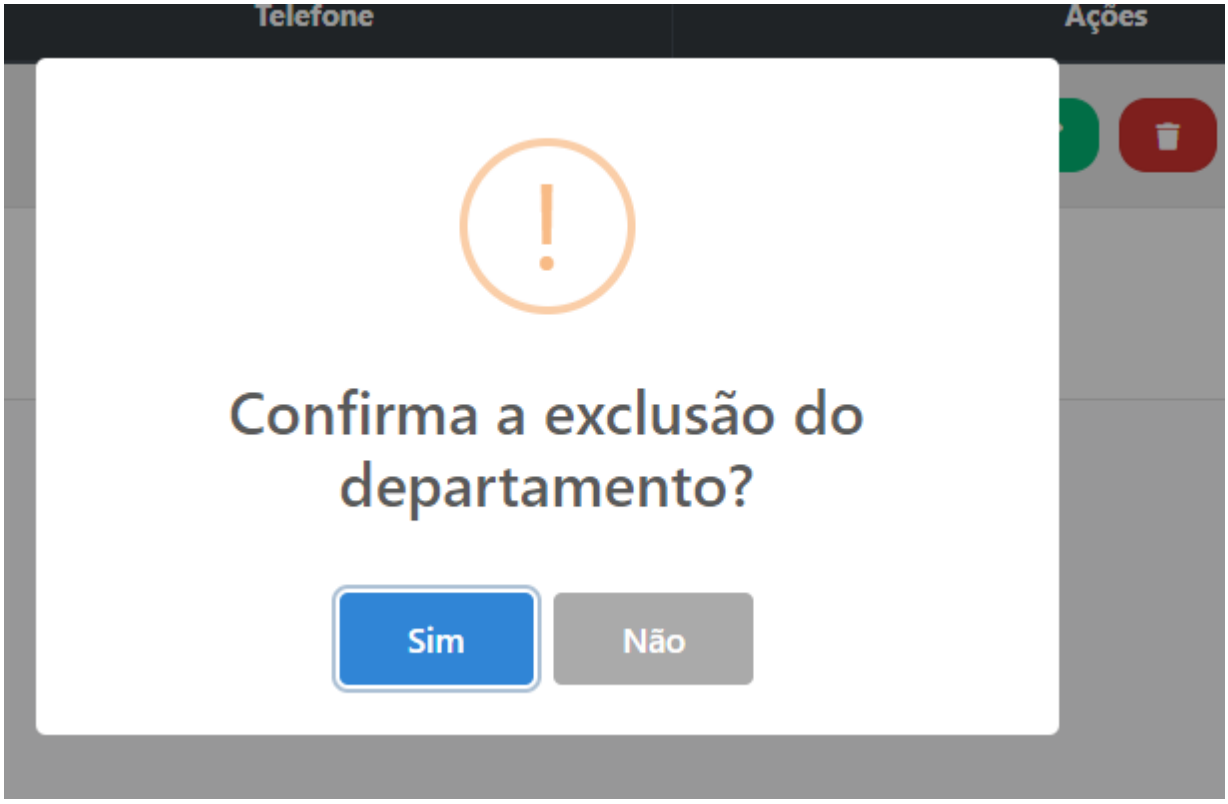


Figura 6.3: Dialog para exclusão do departamento

Codificamos também um botão para navegar até o painel.

```
<a [routerLink]="['/admin/painel']" class="btn btn-primary ">
 <i class="fa fa-search" aria-hidden="true"></i> Voltar</a>
```

Figura 6.4: Botão voltar

Usamos as classes `btn` e `btn-primary` para estilizar o link, passando a rota em `[routerLink]="['/admin/painel']"` .

Continuando a codificação no template do componente `departamento`, vamos escrever o *dialog* com o formulário para capturar os dados, tanto para inclusão como edição dos departamentos.

No trecho a seguir está o início dessa implementação:

```
<p-dialog header="Dados do departamento" [minWidth]="600" [contentStyle]="
{'overflow':'visible'}"   [(visible)]="displayDialogDepartamento"
[responsive]="true" [modal]="true">
  <div class="ui-grid ui-grid-responsive ui-fluid" *ngIf="form.value">
```

Com a tag `<p-dialog` iniciamos a implementação do *dialog*, informando algumas propriedades relacionadas ao tipo `modal`, que se comporta com responsividade `responsive`.

Atribuímos a variável `displayDialogDepartamento` à propriedade `[(visible)]` que declaramos na classe.

Na próxima `div` controlamos a exibição com a diretiva condicional `*ngIf` para evitar erros na renderização dos componentes.

Em seguida, escrevemos o formulário:

```
<form [formGroup]="form">
      <div class="ui-grid-row mb-3">
        <div class="ui-grid-col-4">
          <label for="ocorrencia">Nome*:</label>
        </div>
        <div class="ui-grid-col-8 ">
          <input type="text" pInputText formControlName="nome"  />
        </div>
      </div>
      <div class="ui-grid-row mb-3">
        <div class="ui-grid-col-4">
          <label for="ocorrencia">Telefone:</label>
        </div>
        <div class="ui-grid-col-8 ">
          <p-inputMask formControlName="telefone" mask="(99)9999-9999">
</p-inputMask>
        </div>
```

```
    </div>
  </form>
```

Associamos o formulário ao `FormGroup` da classe com `[formGroup]="form"` , em seguida, definimos o rótulo e o *input*, estilizando com o atributo `pInputText` da biblioteca **PrimeNg**.

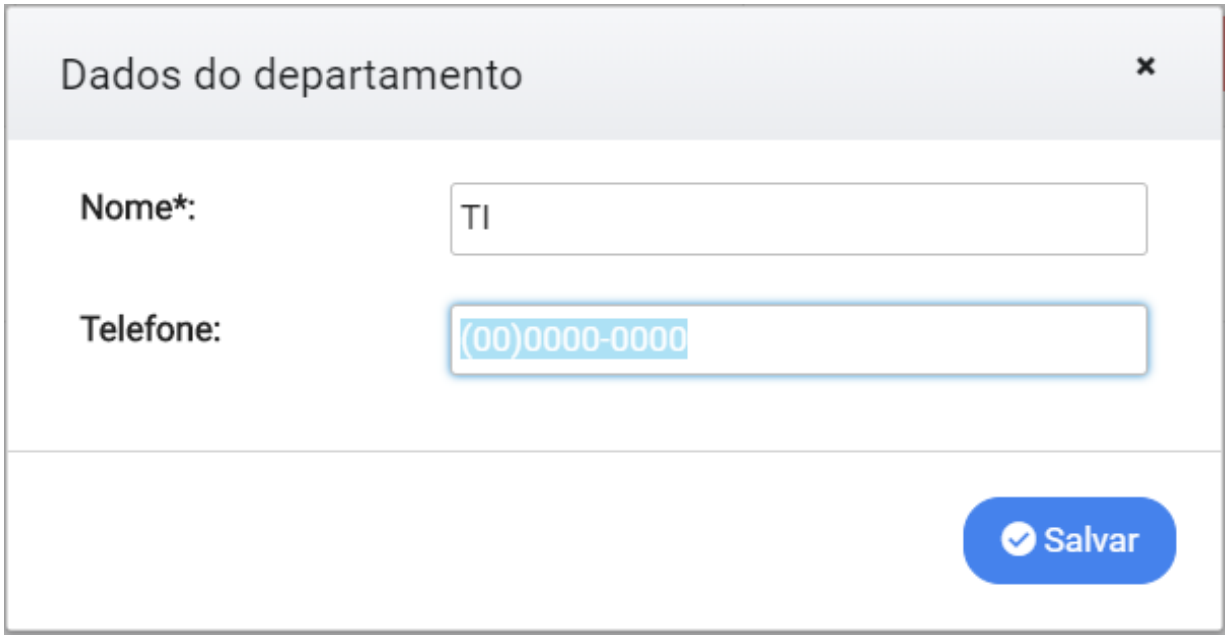


Figura 6.5: Formulário de departamentos

Conectamos os campos do formulário no template `formControlName` com os valores já definidos na classe, como `nome` e `telefone` .

Para o campo telefone, utilizamos um input personalizado, `p-inputMask` , onde registramos uma máscara `mask="(99)9999-9999"` na entrada de valores.

Finalmente, definimos um botão para salvar ou atualizar o registro:

```
 <button [disabled]="!form.valid" type="button" class="btn btn-primary"
(click)="save()">
  <i class="fas fa-check-circle"></i> {{edit ? 'Atualizar' : 'Salvar'}}
</button>
```

O botão fica disponível somente se o estado do formulário for válido `[disabled]="!form.valid"` .

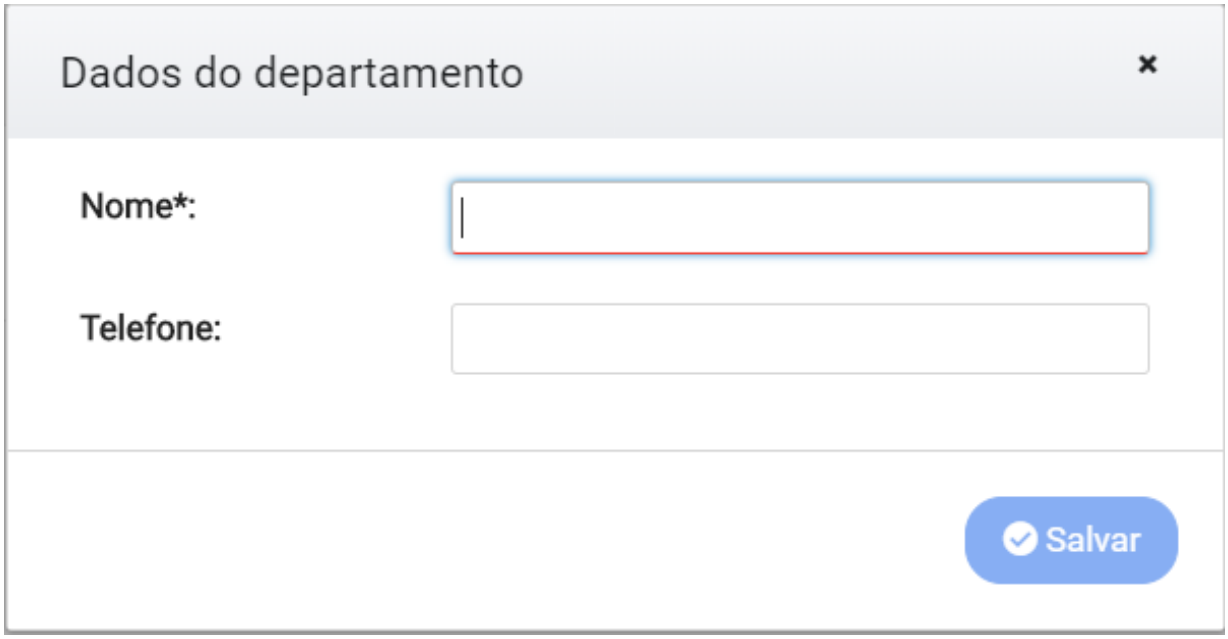


Figura 6.6: Campo com preenchimento obrigatório

Vinculamos ao evento `click` o método que persiste ou altera as informações em `save()`. A exibição do rótulo do botão também verifica em qual modo o formulário se encontra, `{{edit ? 'Atualizar' : 'Salvar'}}`.

Com o preenchimento do campo obrigatório, temos a confirmação da operação no Firebase.

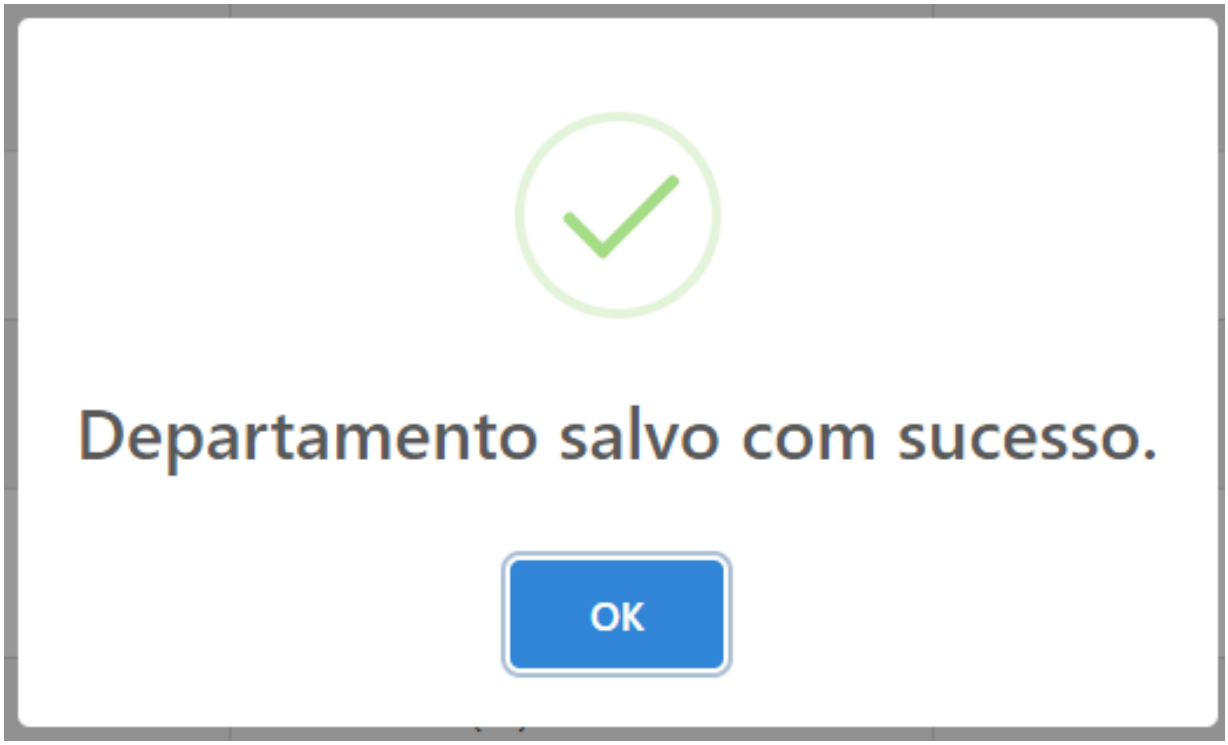


Figura 6.7: Confirmação do departamento salvo

No arquivo de estilização dos componentes `departamento.component.css` , codificamos assim:

```
.ui-grid-row div {
  padding: 4px 10px
}

.ui-grid-row div label {
  font-weight: bold;
}

.col-centered{
  float: none;
  margin: 0 auto;
}

.btn{
  margin: 5px;
}
```

Sobrescrevemos a classe `ui-grid-row` para aumentar o espaçamento interno dos componentes do formulário e negritamos os `label`. Também definimos uma classe para centralizar colunas em `col-centered`.

Por fim, especificamos uma margem de 5 pixels para a classe `btn`, utilizada nos botões.

Após a inclusão de registro é possível visualizar no console do Firebase https://console.firebase.google.com a estrutura criada:

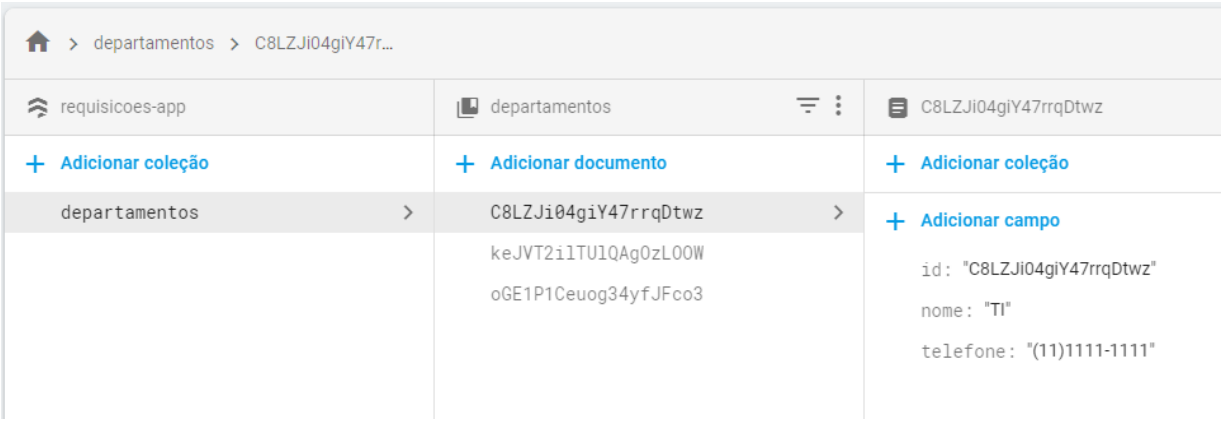


Figura 6.8: Coleção de documentos no Firebase

No primeiro painel à esquerda, temos a coleção `departamentos`. O painel central apresenta os documentos identificados pela chave. E finalmente, no painel da direita, os dados armazenados para os campos `id`, `nome` e `telefone`.

Assim, finalizamos o desenvolvimento do requisito **Cadastrar Departamento**. Abordamos questões de requisitos não funcionais com o uso de *Lazy Loading* além de diferentes formas para trabalhar com formulários.

6.3 Requisito Cadastrar Funcionário

Continuando o desenvolvimento da aplicação, vamos implementar o requisito *Cadastrar Funcionário*. Esse possui algumas particularidades em relação ao requisito anterior.

No final da seção teremos o componente:

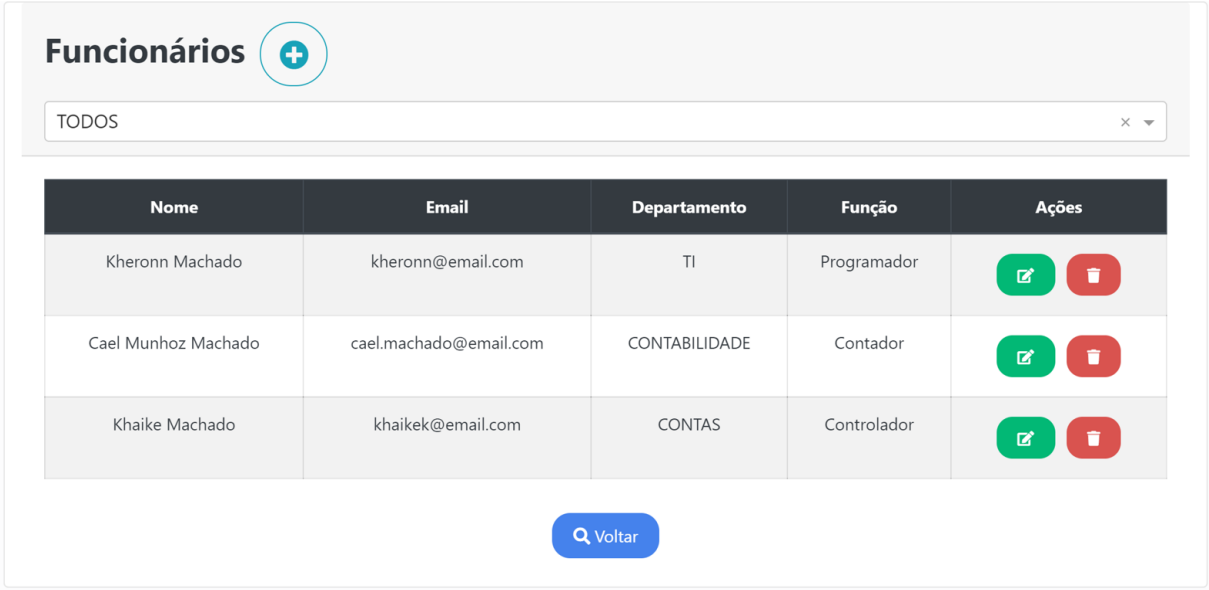


Figura 6.9: Componente funcionario

Começamos gerando o módulo e o componente do funcionário, digitando no terminal os comandos:

```
ng g m components/admin/funcionario --routing
ng g c components/admin/funcionario
```

Assim como no departamento, a mesma estrutura de pasta e arquivos foi gerada. Antes de codificar as classes, vamos utilizar um componente para exibir um combo dos departamentos.

Ainda no terminal, informe:

```
npm install --save @ng-select/ng-select
```

Esse componente fornece várias propriedades e métodos que facilitam as operações com campos do tipo *dropdowns*. Para

configurá-lo, devemos registrar seu módulo e definir uma configuração de estilos.

No módulo do componente funcionário (`funcionario.module.ts`), registramos o módulo e os outros módulos comuns que já deixamos organizados:

```
import { ComumModule } from './../../../modules/comum.module';
import { NgModule } from '@angular/core';
import { NgSelectModule } from '@ng-select/ng-select';
import { FuncionarioRoutingModule } from './funcionario-routing.module';
import { FuncionarioComponent } from './funcionario.component';

@NgModule({
  declarations: [FuncionarioComponent],
  imports: [
    ComumModule,
    FuncionarioRoutingModule,
    NgSelectModule
  ]
})
export class FuncionarioModule { }
```

Temos a declaração de todos os *imports* e a declaração no array de `imports` nos módulos, assim como no componente anterior. Porém, desta vez, registramos o módulo `NgSelectModule` para utilizarmos o componente instalado.

No arquivo de estilos `styles.css` , localizado na raiz do projeto, adicionamos a linha:

```
@import "~@ng-select/ng-select/themes/default.theme.css";
```

Sem essa adição não há exibição do componente. Com isso, temos a configuração do componente da biblioteca completa.

O próximo passo é associar um path ao componente no arquivo de rotas `funcionario-routing.module.ts` .

```
{path:'', component: FuncionarioComponent}
```

E no arquivo de rotas da aplicação `app-routing.module.ts`, adicionamos o `path` para funcionário:

```
  { path: 'admin/funcionario', loadChildren: () =>
import('./components/admin/funcionario/funcionario.module')
  .then(m => m.FuncionarioModule), canActivate: [AuthguardService]},
```

Novamente, iniciamos a codificação mais densa pela classe dos componentes `funcionario.component.ts`, definindo atributos e métodos:

```
export class FuncionarioComponent implements OnInit {

  funcionarios$: Observable<Funcionario[]>;
  departamentos$: Observable<Departamento[]>;
  departamentoFiltro: string;
  edit: boolean;
  displayDialogFuncionario: boolean;
  form: FormGroup;
```

Logo abaixo da assinatura da classe `FuncionarioComponent`, com exceção de `departamentos$` e `departamentoFiltro`, temos os mesmos objetos do componente `departamento`.

O que vai mudar na implementação será a necessidade de exibirmos todos os departamentos para o usuário alocar o funcionário. Além disso, vamos desenvolver um filtro na tabela por departamento, daí a necessidade de controlar o valor do filtro em `departamentoFiltro`.

Continuando o código:

```
 constructor(private funcionarioService: FuncionarioService, private
departamentoService: DepartamentoService, private fb: FormBuilder) { }

 ngOnInit() {
   this.funcionarios$ = this.funcionarioService.list();
   this.departamentos$ = this.departamentoService.list();
   this.departamentoFiltro = 'TODOS'
```

```
    this.configForm();
  }
```

Injetamos no construtor os dois serviços `FuncionarioService` e `DepartamentoService`, além da classe que já utilizamos para o formulário `FormBuilder`.

Em `ngOnInit`, carregamos ambos funcionários e departamentos dos serviços de `list()`, para exibir no template. Também definimos um valor de inicialização para o filtro como `TODOS`. Posteriormente, na implementação do filtro, isso indicará para exibir os funcionários de todos os departamentos.

Também chamamos o método `configForm()`:

```
  configForm() {
    this.form = this.fb.group({
      id: new FormControl(),
      nome: new FormControl('', Validators.required),
      email: new FormControl('', [Validators.required, Validators.email]),
      funcao: new FormControl(''),
      departamento: new FormControl('', Validators.required)
    })
  }
```

Declaramos os campos `id`, `nome`, `email`, `funcao` e `departamento` para o objeto que representa o formulário, instanciando o respectivo controle e validações.

Destaco a possibilidade de informar mais de uma validação, como em `[Validators.required, Validators.email]` para o e-mail.

O restante da implementação é igual ao que já fizemos com o componente departamento, não sendo necessário dar mais detalhes ou explicações.

Para ver o código completo da classe do funcionário, acesse https://bit.ly/2HkYW5J.

6.4 Pipe - Filtrando os registros de funcionários

Antes de codificar o template do componente funcionário, vamos gerar um pipe para filtrar os registros com base na seleção de um combo de departamentos.

Para isso, utilizamos novamente o Angular CLI. No terminal, informe:

```
ng g pipe pipes/filter-departamento
```

Foi gerada uma pasta `pipes` e um arquivo `filter-departamento.pipe.ts`. No código da classe temos inicialmente:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
 name: 'filter'
})
export class FilterDepartamentoPipe implements PipeTransform {
```

Temos a diretiva `@Pipe` informando que a classe é um pipe e um atributo `name` para informar no template.

A classe implementa a interface `PipeTransform`, que tem o método `transform`. Vamos codificá-lo a seguir:

```
transform(value: any, filtro: any): any {
   if (filtro == 'TODOS') return value;
   if (value) {
      return value.filter(elem => (elem.departamento.nome === filtro))
   }
 }
```

O primeiro parâmetro `value` representa o array que será o valor original. Nesse caso, a lista de funcionários. O segundo parâmetro `filtro` será o departamento no qual queremos ver os registros de funcionários.

No primeiro `if`, verificamos se o valor `TODOS` e retornamos a lista sem filtro. No segundo `if`, verificamos se existe uma lista `value` e, em seguida, invocamos o método `filter` comparando cada nome do objeto `departamento` com o valor do filtro `elem.departamento.nome === filtro`.

Com o pipe, já podemos desenvolver o template. Abrindo o código em `funcionario.component.html`, temos inicialmente:

```
<div class="card-header">
     <h3> Funcionários
       <button type="button" style="margin-right: 0px" (click)="add()"
class="text-right btn btn-outline-info btn-lg">
         <i class="fa fa-plus-circle" aria-hidden="true"></i>
       </button>
     </h3>
     <ng-select [(ngModel)]="departamentoFiltro">
       <ng-option [value]="'TODOS'">TODOS</ng-option>
       <ng-option *ngFor="let departamento of departamentos$ | async"
[value]="departamento.nome">{{departamento.nome}}
       </ng-option>
     </ng-select>
```

Seguindo nosso padrão de layout, temos um título com o botão para chamar a função `add()`, igual ao componente `departamento`.

A seguir, temos o código do componente responsável pelo combo `<ng-select`, que exibirá todos os departamentos selecionados pelo atributo `departamentoFiltro`.

Adicionamos uma opção `<ng-option` com o primeiro valor fixo para `TODOS`. No segundo `ng-option` fazemos uma interação `*ngFor="let departamento of departamentos$ | async"` setando tanto o valor quanto o rótulo para `departamento.nome`.

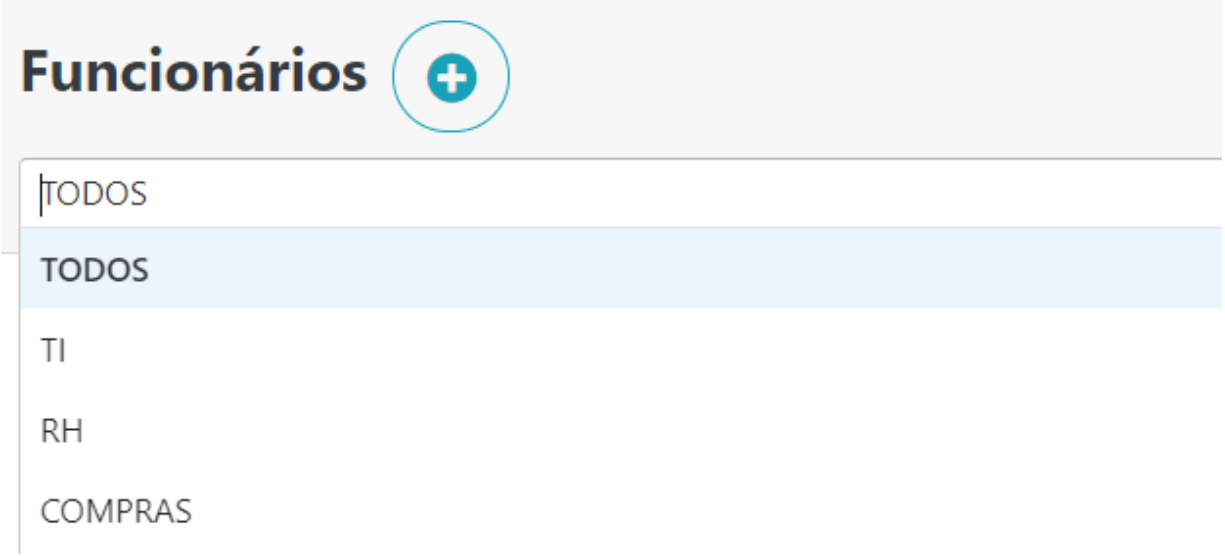


Figura 6.10: Combo departamentos

Seguindo, temos o código para a tabela de funcionários:

```
<table class="table table-striped table-hover table-bordered col-
centered">
       <thead class="thead-dark">
         <tr>
           <th class="text-center">Nome</th>
           <th class="text-center">Email</th>
           <th class="text-center">Departamento</th>
           <th class="text-center">Função</th>
           <th class="text-center">Ações</th>
         </tr>
       </thead>
       <tbody>
         <tr *ngFor="let funcionario of funcionarios$ | async | filter :
departamentoFiltro">
           <td class="text-center">{{funcionario.nome}}</td>
           <td class="text-center">{{funcionario.email}}</td>
           <td class="text-center">{{funcionario.departamento.nome}}</td>
           <td class="text-center">{{funcionario.funcao}}</td>
           <td class="text-center">
             <button type="button"
(click)="selecionaFuncionario(funcionario)" class="btn btn-success  ">
               <i class="fas fa-edit"></i>
             </button>
```

```
            <button type="button" (click)="delete(funcionario)"
class="btn btn-danger  ">
              <i class="fas fa-trash"></i>
            </button>
          </td>
        </tr>
      </tbody>
    </table>
```

Temos as definições de cabeçalho para os campos `nome`, `email`, `departamento` e `funcao`, e para os botões de ações.

Na linha do `ngFor` está o pipe para listas assíncronas `| async` e o pipe personalizado que desenvolvemos para filtrar os resultados com `| filter : departamentoFiltro`, indicando o parâmetro que será o filtro após os `:`.

A seguir codificamos o *dialog* que contém o formulário de funcionário:

```
<p-dialog header="Dados do funcionário" [minWidth]="600" [contentStyle]="
{'overflow':'visible'}"
 [(visible)]="displayDialogFuncionario" [responsive]="true"
[modal]="true">
 <div class="ui-grid ui-grid-responsive ui-fluid" *ngIf="form.value">
   <form [formGroup]="form">
     <div class="ui-grid-row mb-3">
       <div class="ui-grid-col-4">
         <label for="ocorrencia">Nome*:</label>
       </div>
       <div class="ui-grid-col-8 ">
         <input type="text" pInputText formControlName="nome" />
       </div>
     </div>
     <div class="ui-grid-row mb-3">
       <div class="ui-grid-col-4">
         <label for="ocorrencia">Email*:</label>
       </div>
       <div class="ui-grid-col-8 ">
         <input type="text" pInputText formControlName="email" />
       </div>
```

```
      </div>
      <div class="ui-grid-row mb-3">
        <div class="ui-grid-col-4">
          <label for="ocorrencia">Departamento*:</label>
        </div>
        <div class="ui-grid-col-8 ">
          <ng-select [items]="departamentos$ | async" bindLabel="nome"
formControlName="departamento">
          </ng-select>
        </div>
      </div>
      <div class="ui-grid-row mb-3">
        <div class="ui-grid-col-4">
          <label for="ocorrencia">Função:</label>
        </div>
        <div class="ui-grid-col-8 ">
          <input type="text" pInputText formControlName="funcao" />
        </div>
      </div>
    </form>
  </div>
  <p-footer>
<div *ngIf="form.controls['nome'].errors || form.controls['email'].errors
|| form.controls['departamento'].errors "class="text-warning text-left
">*Preenchimento Obrigatório</div>
    <div class="ui-dialog-buttonpane ">
      <button [disabled]="!form.valid" type="button" class="btn btn-
primary" (click)="save()">
        <i class="fas fa-check-circle"></i> {{edit ? 'Atualizar' :
'Salvar'}}
      </button>
    </div>
  </p-footer>
</p-dialog>
```

Temos a definição dos campos de entrada `nome` , `email` , e `funcao` . Para o departamento, utilizamos novamente o componente `ng-select` , porém, como só precisamos trazer os valores do Firebase, conseguimos abreviar a escrita informando o atributo da classe com a propriedade `[items]="departamentos$ | async"` .

Também definimos uma mensagem que será exibida com base na validação dos campos, usando o operador `ngIf` e as condições dos campos `nome`, `email` e `departamento` em `form.controls['nome'].errors || form.controls['email'].errors || form.controls['departamento'].errors`.

O código completo do template está disponível em https://bit.ly/2YrRyfN.

Para finalizar esse requisito, definimos um estilo adicional para controlar o estado do formulário. No arquivo `funcionario.component.css` codificamos a seguinte classe:

```
.ng-invalid:not(form) {
 background-color: rgb(235, 185, 185);
}
```

Assim, definimos uma cor de fundo para campos que ainda não estão validados. O visual do *dialog* ficou assim:

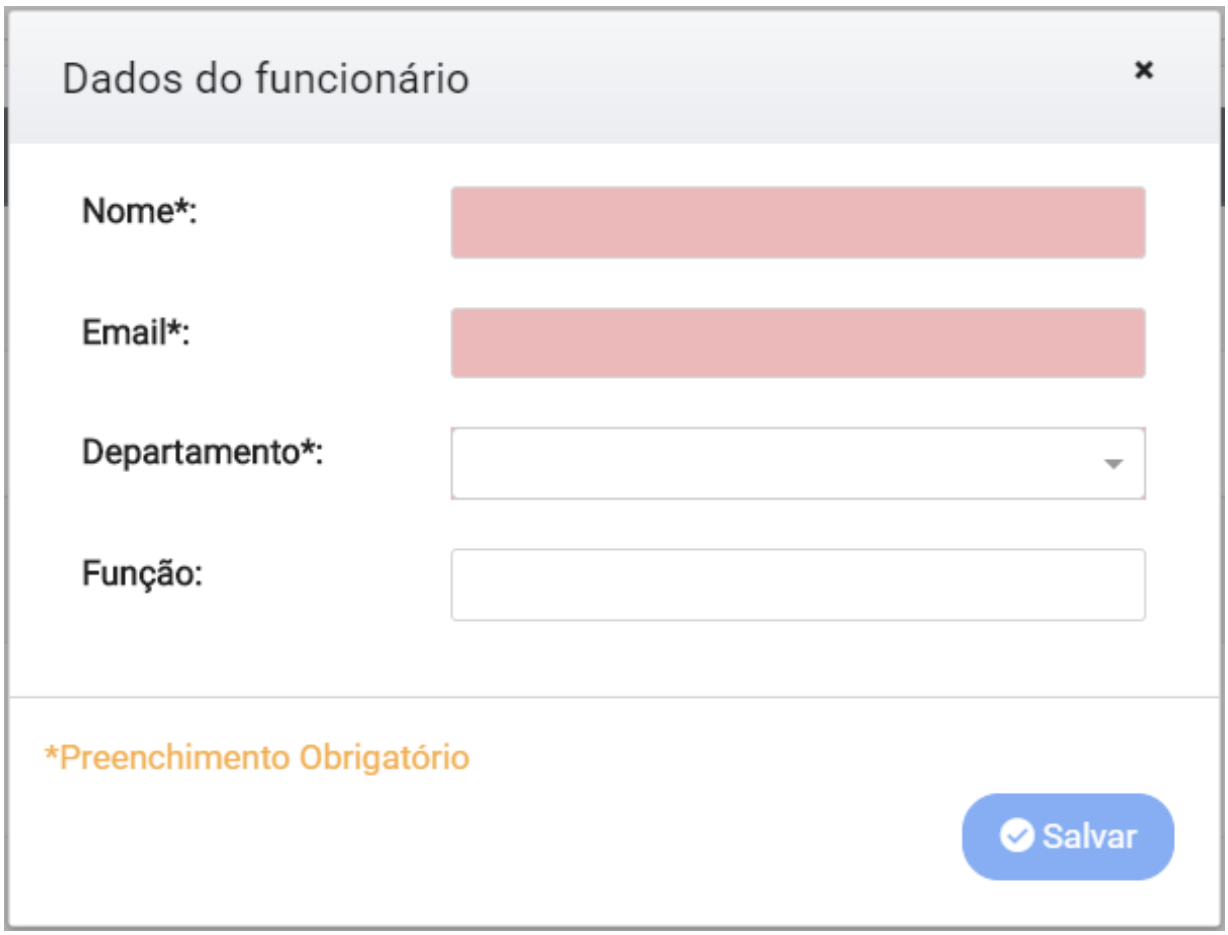


Figura 6.11: Dialog funcionário

Finalizamos o desenvolvimento do requisito *Cadastrar Funcionário*. Conseguimos explorar outras nuances da validação e a construção de pipe na formatação de valores com base em condições.

A seguir, codificaremos o requisito *Gerenciar Requisições*.

Capítulo 7
Mais componentes - Requisito Gerenciar Requisições

Neste capítulo, desenvolveremos o requisito *Gerenciar Requisições*, que consiste em criar funções e telas para que o funcionário possa fazer uma requisição e também gerenciar movimentações de uma requisição solicitada ao seu departamento.

Vamos explorar conceitos-chaves do framework como **@Input** e **@Output** no desenvolvimento de componentes reutilizáveis.

O capítulo será dividido em três seções. Na primeira, desenvolveremos o componente para criar as requisições. Na segunda, um componente para dar seguimentos através de movimentações da requisição; e no terceiro, um componente para visualizar e editar todas as movimentações do requisito.

No final do capítulo teremos a seguinte composição do componente.

Figura 7.1: Gerenciar Requisições

7.1 Minhas Requisições

O primeiro componente será responsável pela inclusão da requisição. Assim, devemos escrever uma função que recupere os dados do funcionário com base no usuário logado.

No arquivo de serviços do funcionário, `funcionario.service.ts`, vamos incluir o método:

```
getFuncionarioLogado(email: string) {
   return this.firestore.collection<Funcionario>('funcionarios', ref =>
     ref.where('email', '==', email)
   ).valueChanges()
 }
```

Passamos o `email` e, através da referência `ref`, criamos uma consulta na coleção de funcionários em `where('email', '==', email)`.

Com isso, podemos criar o componente. No terminal, informe:

```
ng g m components/admin/requisicao --routing
ng g c components/admin/requisicao
```

Inserimos mais uma rota do módulo em `app-routing.module.ts`:

```
  { path: 'admin/requisicao', loadChildren: () =>
import('./components/admin/requisicao/requisicao.module').then(m =>
m.RequisicaoModule), canActivate: [AuthguardService]},
```

E no menu da aplicação indicamos a rota no link em:

```
<li class="nav-item" routerLinkActive="active">
     <a class="nav-link" routerLink="/admin/requisicao" href="#">
    <i class="fas fa-calendar-week"></i>Requisições</a>
</li>
```

No módulo de requisição `requisicao.module.ts`, realizamos os *imports* já conhecidos:

```
import { ComumModule } from 'src/app/modules/comum.module';
import { NgModule } from '@angular/core';
import { RequisicaoRoutingModule } from './requisicao-routing.module';
import { RequisicaoComponent } from './requisicao.component';
import { NgSelectModule } from '@ng-select/ng-select';

@NgModule({
 declarations: [RequisicaoComponent],
 imports: [
   ComumModule,
    RequisicaoRoutingModule,
    NgSelectModule
 ]
})
export class RequisicaoModule { }
```

Na classe do componente `requisicao.component.ts` iniciamos pela declaração dos atributos que utilizaremos:

```
export class RequisicaoComponent implements OnInit {

 requisicoes$: Observable<Requisicao[]>;
 departamentos$: Observable<Departamento[]>;
 edit: boolean;
 displayDialogRequisicao: boolean;
 form: FormGroup;
 funcionarioLogado: Funcionario;
```

Além dos objetos para inserção e exibição nos componentes, vamos precisar consultar o funcionário que está logado, `funcionarioLogado`. Também precisaremos de uma lista de departamentos para indicar o destino da requisição usando `departamentos$`.

Injetamos as classes de serviços e formulários em:

```
constructor(private requisicaoService: RequisicaoService,
    private departamentoService: DepartamentoService,
    private auth: AuthenticationService,
    private funcionarioService: FuncionarioService,
    private fb: FormBuilder) { }
```

Incluímos também a classe de autenticação `AuthenticationService` para recuperarmos o usuário autenticado.

A seguir, escrevemos o método para recuperar o funcionário:

```
 async recuperaFuncionario() {
   await this.auth.authUser()
     .subscribe(dados => {
       this.funcionarioService.getFuncionarioLogado(dados.email)
         .subscribe(funcionarios => {
           this.funcionarioLogado = funcionarios[0];
            this.requisicoes$ = this.requisicaoService.list()
              .pipe(
                map((reqs: Requisicao[]) => reqs.filter(r =>
r.solicitante.email === this.funcionarioLogado.email))
              )
         })
     })
 }
```

Chamamos o método `getFuncionarioLogado` passando o email do método `authUser`.

Na sequência, recuperamos o usuário na primeira posição do array, atribuindo este ao funcionário, em `this.funcionarioLogado = funcionarios[0]`.

O objetivo é mostrar somente as requisições incluídas pelo usuário logado. Realizamos esse filtro através do método `filter` no parâmetro do callback do array de `requisicoes$`.

Por último, comparamos as propriedades email do solicitante da requisição com o do funcionário logado em `r.solicitante.email === this.funcionarioLogado.email`.

No próximo método, configuramos o formulário:

```
  configForm() {
    this.form = this.fb.group({
      id: new FormControl(),
      destino: new FormControl('', Validators.required),
      solicitante: new FormControl(''),
      dataAbertura: new FormControl(''),
      ultimaAtualizacao: new FormControl(''),
      status: new FormControl(''),
      descricao: new FormControl('', Validators.required)
    })
  }
```

Definimos os campos `id`, `destino`, `solicitante`, `dataAbertura`, `ultimaAtualizacao`, `status` e `descricao`, indicando quais terão validações. A diferença dos últimos componentes fica na implementação do método `add()` que invoca o método `setValorPadrao()`.

```
add() {
   this.form.reset();
   this.edit = false;
   this.displayDialogRequisicao = true;
```

```
    this.setValorPadrao();
  }
```

Precisamos setar valores padrões na abertura de toda requisição.

```
  setValorPadrao() {
    this.form.patchValue({
      solicitante: this.funcionarioLogado,
      status: 'aberto',
      dataAbertura: new Date(),
      ultimaAtualizacao: new Date()
    })
  }
```

Utilizamos o método `patchValue` para fazer uma atualização parcial nos campos do formulário, informando o funcionário logado, o status para `aberto` e a data atual para os campos `dataAbertura` e `ultimaAtualizacao` com uma nova instância, através do `new Date()`.

Os métodos para salvar, recuperar e excluir são iguais aos componentes do capítulo anterior.

Para ver o código completo da classe, acesse https://bit.ly/2JyE912.

O próximo passo é a codificação do template em `requisicao.component.html`.

No trecho em que codificamos a tabela:

```
<table class="table table-striped table-hover table-bordered col-
centered">
        <thead class="thead-dark">
          <tr>
            <th class="text-center">Abertura</th>
            <th class="text-center">Última atualização</th>
            <th class="text-center">Departamento</th>
            <th class="text-center">Status</th>
            <th class="text-center">Movimentações</th>
            <th class="text-center">Ações</th>
          </tr>
        </thead>
```

```
        <tbody>
          <tr *ngFor="let requisicao of requisicoes$ | async ">
            <td class="text-center">{{requisicao.dataAbertura.toDate() |
date : 'dd/MM/yyyy HH:mm'}}</td>
            <td class="text-center">{{requisicao.ultimaAtualizacao.toDate()
| date : 'dd/MM/yyyy HH:mm'}}</td>
            <td class="text-center">{{requisicao.destino.nome}}</td>
            <td class="text-center">{{requisicao.status}}</td>
            <td class="text-center">
              <span class="badge badge-pill badge-secondary">
                {{!requisicao.movimentacoes?.length ? '0':
requisicao.movimentacoes?.length}}
              </span>
            </td>
            <td class="text-center">
              <button type="button"
(click)="selecionaRequisicao(requisicao)" class="btn btn-success  ">
                <i class="fas fa-edit"></i>
              </button>
              <button type="button" (click)="delete(requisicao)" class="btn
btn-danger  ">
                <i class="fas fa-trash"></i>
              </button>
            </td>
          </tr>
        </tbody>
      </table>
```

Definimos o cabeçalho da tabela com os campos da requisição. Em seguida, utilizamos um pipe para exibir campos de data.

Para isso, devemos chamar o método `toDate()` do atributo e, em seguida, usar o operador e o pipe `| date`. Definimos ainda o formato da data e hora após os dois pontos em `dd/MM/yyyy HH:mm`.

Também informamos o número de movimentações da requisição, através do tamanho do array em `requisicao.movimentacoes?.length`. Usamos o operador safe `?` para evitar exceções de valores nulos ou indefinidos.

No detalhe, o resultado da formatação para cada linha:

Figura 7.2: Registro da Requisição

O *dialog* que contém o formulário segue o padrão dos componentes de inclusão:

```
<p-dialog header="Dados da Requisição" [minWidth]="600" [contentStyle]="
{'overflow':'visible'}"
 [(visible)]="displayDialogRequisicao" [responsive]="true" [modal]="true">
 <div class="ui-grid ui-grid-responsive ui-fluid" *ngIf="form.value">
   <form [formGroup]="form">
     <div class="ui-grid-row mb-3">
       <div class="ui-grid-col-4">
         <label for="departamento">Destino*:</label>
       </div>
       <div class="ui-grid-col-8 ">
         <ng-select [items]="departamentos$ | async" bindLabel="nome"
formControlName="destino">
         </ng-select>
       </div>
     </div>
     <div class="ui-grid-row mb-3">
       <div class="ui-grid-col-4">
         <label for="descricao">Descricao*:</label>
       </div>
       <div class="ui-grid-col-8 ">
         <textarea rows="5" cols="30" pInputTextarea
formControlName="descricao"></textarea>
       </div>
     </div>
   </form>
 </div>
 <p-footer>
   <div *ngIf="form.controls['destino'].errors ||
form.controls['descricao'].errors" class="text-warning text-left ">
```

```
      *Preenchimento Obrigatório</div>
    <div class="ui-dialog-buttonpane ">
      <button [disabled]="!form.valid" type="button" class="btn btn-
primary" (click)="save()">
        <i class="fas fa-check-circle"></i> {{edit ? 'Atualizar' :
'Salvar'}}
      </button>
    </div>
  </p-footer>
</p-dialog>
```

A diferença está no uso do campo `textarea` para o atributo `descricao`.

O resultado do *dialog* pode ser visto aqui:

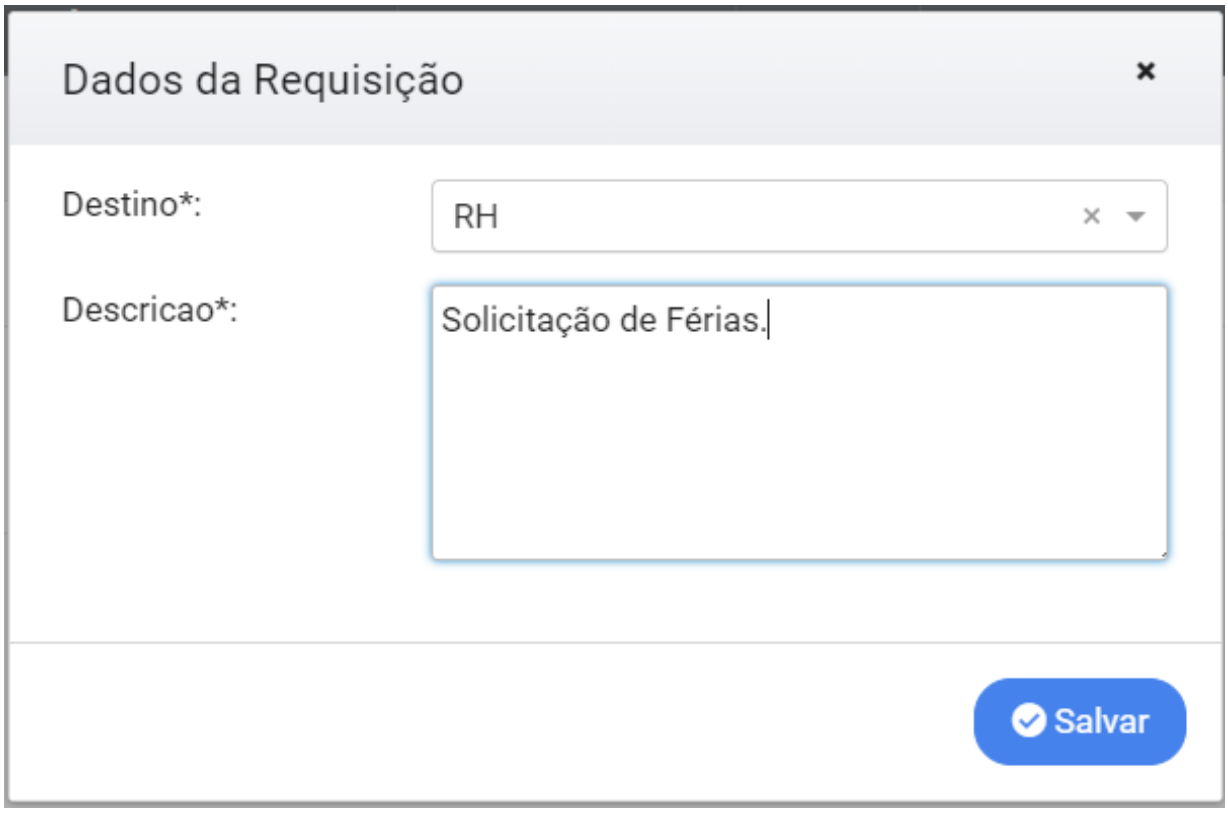


Figura 7.3: Dialog da Requisição

Abrindo o console do Firebase (https://console.firebase.google.com), no menu `Database`, temos as coleções armazenadas com os

respectivos documentos.

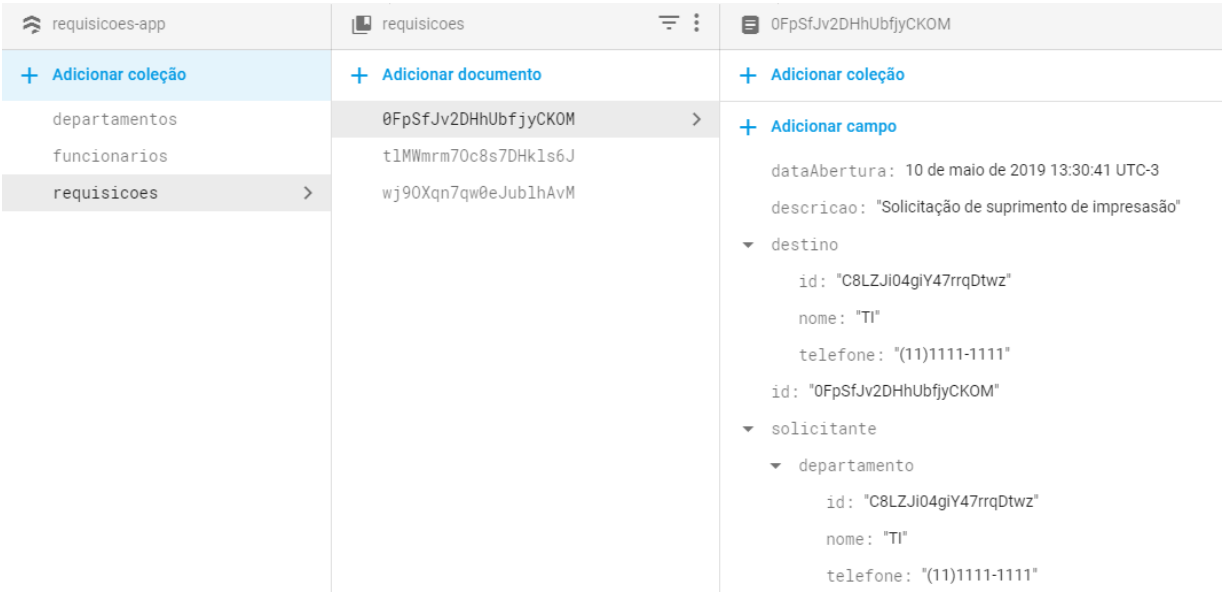


Figura 7.4: Coleções no Firebase

Para visualizar a implementação completa do componente, acesse https://bit.ly/2YpgZyg.

Assim, finalizamos a primeira parte do requisito. Na próxima seção vamos construir o componente para dar movimentações às requisições.

7.2 Requisições solicitadas - Trabalhando com @Input

No próximo componente, vamos listar todas as requisições destinadas ao departamento do funcionário logado.

O objetivo também será criar um componente reutilizável em outra parte da aplicação, no caso, o painel.

Assim, vamos explorar o uso do `@Input` para criar interação entre os componentes.

No terminal, criamos o módulo e o componente `movimentacao` informando:

```
ng g m components/admin/movimentacao --routing
ng g c components/admin/movimentacao
```

Começamos importando os módulos básicos para manipulação e exibição de dados no componente em `movimentacao.module.ts` na pasta `movimentacao` :

```
import { NgModule } from '@angular/core';
import { MovimentacaoRoutingModule } from './movimentacao-routing.module';
import { MovimentacaoComponent } from './movimentacao.component';
import { ComumModule } from 'src/app/modules/comum.module';
import { NgSelectModule } from '@ng-select/ng-select';

@NgModule({
 declarations: [MovimentacaoComponent],
 imports: [
   ComumModule,
   MovimentacaoRoutingModule,
   NgSelectModule
 ]
})
export class MovimentacaoModule { }
```

E a configuração da rota para utilizarmos o *Lazy Loading*, no arquivo `movimentacao-routing.module.ts` :

```
const routes: Routes = [
 { path: '', component: MovimentacaoComponent }
];
```

Não vamos criar uma rota específica para esse componente. Isso porque utilizaremos uma abordagem de componente *pai-filho*. Nesse caso, a classe pai será o componente da requisição e vamos referenciar no template `requisicao.component.html` o seletor da movimentação `app-movimentacao` disponível na classe do componente `movimentacao.component.ts` .

Em seguida, começamos a codificação dessa classe. Criaremos os atributos e métodos para incluir movimentações na requisição, assim temos os seguintes campos abaixo da assinatura da classe:

```
export class MovimentacaoComponent implements OnInit {
 @Input() funcionarioLogado: Funcionario;
 requisicoes$: Observable<Requisicao[]>;
 movimentacoes: Movimentacao[];
 requisicaoSelecionada: Requisicao;
 edit: boolean;
 displayDialogMovimentacao: boolean;
 displayDialogMovimentacoes: boolean;
 form: FormGroup;
 listaStatus: string[];
```

Conforme mencionado, faremos uso do `@Input()` associando ao atributo `funcionarioLogado` . Dessa forma, indicamos esse atributo como uma propriedade de entrada.

Durante a detecção de alterações, o Angular atualiza as propriedades dos dados com o valor informado no componente *pai*.

Ainda nas declarações, definimos duas listas para exibibir as requisições e as movimentações, através dos objetos `requisicoes$` e `movimentacoes` , além de duas propriedades para exibição de *dialogs*, `displayDialogMovimentacao` e `displayDialogMovimentacoes` .

Finalizamos a declaração com um atributo para exibir as opções de status para a movimentação, `listaStatus` , com um array de string.

Continuando a escrita do código, vamos invocar em `constructor` alguns métodos para definir os campos do formulário, listar as requisições e carregar as opções de status.

```
constructor(private requisicaoService: RequisicaoService, private fb:
FormBuilder) { }
 ngOnInit() {
  this.configForm();
   this.carregaStatus();
```

```
    this.listaRequisicoesDepartamento();
  }
```

No método construtor temos o serviço da requisição `RequisicaoService` e a classe para o formulário `FormBuilder`, além dos métodos que invocaremos na inicialização do componente.

A seguir, escrevemos o método `configForm()`:

```
  configForm() {
    this.form = this.fb.group({
      funcionario: new FormControl('', Validators.required),
      dataHora: new FormControl(''),
      status: new FormControl('', Validators.required),
      descricao: new FormControl('', Validators.required)
    })
  }
```

Definimos os campos `funcionario`, `dataHora`, `status` e `descricao` para incluir a requisição, definindo as validações com `Validators.required`.

Na sequência, vamos implementar o método `carregaStatus()`.

```
carregaStatus() {
    this.listaStatus = ['Aberto', 'Pendente', 'Processando', 'Não
autorizada', 'Finalizado'];
  }
```

Populamos o array `listaStatus` com as opções definidas entre os colchetes.

E finalmente temos o método `listaRequisicoesDepartamento()`:

```
listaRequisicoesDepartamento() {
    this.requisicoes$ = this.requisicaoService.list()
      .pipe(
        map((reqs: Requisicao[]) =>
        reqs.filter(r =>
        r.destino.nome === this.funcionarioLogado.departamento.nome))
      )
  }
```

Para listar somente as requisições do departamento do funcionário utilizamos o método `filter`, comparando em cada objeto `r` o nome do departamento: `r.destino.nome === this.funcionarioLogado.departamento.nome`.

Ainda na sequência da classe `movimentacao.component.ts`, implementaremos uma função para adicionar uma nova requisição:

```
add(requisicao: Requisicao) {
   this.form.reset();
   this.edit = false;
   this.setValorPadrao();
   this.requisicaoSelecionada = requisicao;
   this.movimentacoes = (!requisicao.movimentacoes ? [] :
requisicao.movimentacoes);
   this.displayDialogMovimentacao = true;
 }
```

No parâmetro do método, passamos a requisição que será selecionada no template do componente. Em seguida, atribuímos em `this.requisicaoSelecionada` o objeto `requisicao` para manipular posteriormente.

Ainda, associamos o valor de movimentações, verificando antes a existência de valores em `!requisicao.movimentacoes`.

No próximo método, `save()`, efetivamos a persistência das movimentações, atualizando alguns campos da requisição, conforme a implementação a seguir:

```
save() {
    this.movimentacoes.push(this.form.value);
    this.requisicaoSelecionada.movimentacoes = this.movimentacoes;
    this.requisicaoSelecionada.status = this.form.controls['status'].value
    this.requisicaoSelecionada.ultimaAtualizacao = new Date();
    this.requisicaoService.createOrUpdate(this.requisicaoSelecionada)
      .then(() => {
        this.displayDialogMovimentacao = false;
        Swal.fire(`Requisição ${!this.edit ? 'salvo' : 'atualizado'} com
sucesso.`, '', 'success');
```

```
      })
      .catch((erro) => {
        this.displayDialogMovimentacao = true;
        Swal.fire(`Erro ao ${!this.edit ? 'salvo' : 'atualizado'} o
Requisição.`, `Detalhes: ${erro}`, 'error');
      })
    this.form.reset()
  }
```

Inicialmente adicionamos a movimentação no array com os valores do formulário, em `this.movimentacoes.push(this.form.value)`, e atribuímos o array de movimentações ao objeto da requisição selecionada, no trecho `this.requisicaoSelecionada.movimentacoes = this.movimentacoes`.

Em seguida, atualizamos o status e a data da última atualização com `this.requisicaoSelecionada.status = this.form.controls['status'].value` e `this.requisicaoSelecionada.ultimaAtualizacao = new Date()`.

Assim invocamos o método `createOrUpdate` com o objeto `requisicaoSelecionada`.

Vale ressaltar que o método é uma operação de atualização, pois já temos no banco a requisição. O que fizemos foi adicionar elementos no array da propriedade `movimentacoes` do modelo.

O método para excluir é igual às outras implementações.

Para finalizar a classe, vamos incluir dois métodos que servirão basicamente para fechar o *dialog* do próximo componente:

```
onDialogClose(event) {
    this.displayDialogMovimentacoes = event;
  }
```

E para visualizar as movimentações da requisição:

```
 verMovimentacoes(requisicao: Requisicao) {
   this.requisicaoSelecionada = requisicao;
```

```
    this.movimentacoes = requisicao.movimentacoes;
    this.displayDialogMovimentacoes = true;
  }
```

Passamos no parâmetro a requisição selecionada, `verMovimentacoes(requisicao: Requisicao)`, e atribuímos aos objetos `requisicaoSelecionada` e `movimentacoes` o objeto `requisicao`, declarado no parâmetro. Também exibimos o *dialog* de movimentações com `displayDialogMovimentacoes = true`.

Já temos toda a lógica do componente programada. Vamos desenvolver o template em `movimentacao.component.html`, utilizando o padrão de layout assumido na aplicação.

Template do componente Movimentação

Vamos iniciar analisando o trecho responsável pela exibição da tabela, conforme o código a seguir.

```
<table class="table table-striped table-hover table-bordered col-
centered">
       <thead class="thead-dark">
         <tr>
           <th class="text-center">Abertura</th>
           <th class="text-center">Última atualização</th>
           <th class="text-center">Departamento</th>
           <th class="text-center">Status</th>
           <th class="text-center">Movimentações</th>
           <th class="text-center">Ações</th>
         </tr>
       </thead>
       <tbody>
         <tr *ngFor="let requisicao of requisicoes$ | async ">
           <td class="text-center">{{requisicao.dataAbertura?.toDate() |
date : 'dd/MM/yyyy HH:mm'}}</td>
           <td class="text-center">{{requisicao.ultimaAtualizacao?.seconds
* 1000  | date : 'dd/MM/yyyy HH:mm' }}</td>
           <td class="text-center">{{requisicao.destino.nome}}</td>
           <td class="text-center">{{requisicao.status}}</td>
           <td class="text-center">
```

```
            <span class="badge badge-pill badge-secondary">
              {{!requisicao.movimentacoes?.length ? '0':
requisicao.movimentacoes?.length}}
            </span>
          </td>
          <td class="text-center">
            <button type="button" (click)="add(requisicao)" class="btn
btn-info">
              <i class="fa fa-plus-circle"></i>
            </button>
            <button type="button" (click)="verMovimentacoes(requisicao)"
class="btn btn-success">
              <i class="far fa-list-alt"></i>
            </button>
          </td>
        </tr>
      </tbody>
    </table>
```

Temos a definição das colunas na primeira linha `<tr>` para exibição dos valores de cada requisição. Na sequência, iteramos sobre cada objeto, utilizando o operador `*ngFor="let requisicao of requisicoes$` .

Observe uma mudança na exibição de campos do tipo `date` . Até agora usávamos a seguinte notação para exibição desses valores: `objeto.propriedade.toDate() | date` .

Dessa vez codificamos a exibição do campo de última atualização recuperando os segundos e multiplicando por 1000, em `requisicao.ultimaAtualizacao?.seconds * 1000` . Dessa forma conseguimos exibir o campo no formato de data corretamente sem erros ao incluir movimentações ao atualizar o atributo da requisição.

Na definição dos botões, temos o método `add(requisicao)` no evento `click` responsável pela nova movimentação. Declaramos também o método que exibe todas as movimentações em `"verMovimentacoes(requisicao)` .

No código seguinte, fazemos o *dialog* para inclusão da movimentação:

```
<p-dialog header="Dados da Requisição" [minWidth]="600" [contentStyle]="
{'overflow':'visible'}"
  [(visible)]="displayDialogMovimentacao" [responsive]="true"
[modal]="true">
  <div class="ui-grid ui-grid-responsive ui-fluid" *ngIf="form.value">
    <div class="ui-grid-row mb-3">
      <div class="ui-grid-col-4">
        <label for="departamento">Solicitante:</label>
      </div>
      <div class="ui-grid-col-8 text-primary">
        {{requisicaoSelecionada?.solicitante.nome}}
      </div>
    </div>
    <div class="ui-grid-row mb-3">
      <div class="ui-grid-col-4">
        <label for="descricao">Solicitação:</label>
      </div>
      <div class="ui-grid-col-8 ">
        <textarea rows="5" cols="30" disabled pInputTextarea
[value]="requisicaoSelecionada?.descricao"></textarea>
      </div>
    </div>
    <form [formGroup]="form">
      <div class="ui-grid-row mb-3">
        <div class="ui-grid-col-4">
          <label for="departamento">Status*:</label>
        </div>
        <div class="ui-grid-col-8 ">
          <ng-select [items]="listaStatus" formControlName="status">
          </ng-select>
        </div>
      </div>
      <div class="ui-grid-row mb-3">
        <div class="ui-grid-col-4">
          <label for="descricao">Descricao*:</label>
        </div>
        <div class="ui-grid-col-8 ">
         <textarea rows="5" cols="30" pInputTextarea
```

```
formControlName="descricao"></textarea>
        </div>
      </div>
    </form>
  </div>
  <p-footer>
    <div *ngIf="form.controls['status'].errors ||
form.controls['descricao'].errors" class="text-warning text-left ">
      *Preenchimento Obrigatório</div>
    <div class="ui-dialog-buttonpane ">
      <button [disabled]="!form.valid" type="button" class="btn btn-
primary" (click)="save()">
        <i class="fas fa-check-circle"></i> {{edit ? 'Atualizar' :
'Salvar'}}
      </button>
    </div>
  </p-footer>
</p-dialog>
```

No primeiro segmento, `{{requisicaoSelecionada?.solicitante.nome}}`, exibimos o nome do solicitante, e a descrição da solicitação na propriedade com o trecho `[value]="requisicaoSelecionada?.descricao"` do campo de texto, porém, a marcamos como desabilitada, com `disabled`.

Para o objeto `requisicaoSelecionada`, utilizamos novamente o operador *safe* `?` para evitar erros de renderização.

Na sequência, temos a tag `<form` com a declaração dos campos de entrada para `status` e `descricao`.

No campo `status` utilizamos o componente `<ng-select` para exibir as opções com a fonte de dados em `[items]="listaStatus"`.

O restante segue igual aos componentes já desenvolvidos.

Para ver o código completo do componente, acesse https://bit.ly/2WCVUjv.

Dessa forma, já temos o componente construído, ou seja, a lógica e o template, porém, ele não aparece ainda no navegador.

A seguir, vamos associar o componente filho `movimentacao.component.ts` no componente pai `requisicao.component.ts`

7.3 Associando os componentes

Vamos abrir o template `requisicao.component.html` e adicionar a seguinte linha:

```
<app-movimentacao *ngIf="funcionarioLogado"
[funcionarioLogado]="funcionarioLogado"></app-movimentacao>
```

A tag `<app-movimentacao` é responsável pela renderização do componente `movimentacao` recém-criado. Informamos a diretiva condicional com `*ngIf` para o atributo `funcionarioLogado` . Dessa forma, conseguimos evitar erros de renderização.

Usamos o atributo `[funcionarioLogado]` para conectar os componentes *pai-filho*, concluindo a abordagem para reutilizar o componente.

Para finalizar esse componente, devemos informar o componente filho no módulo do pai.

Para isso, abrimos a classe `requisicao.module.ts` e incluímos no array de declaração em `declarations` o componente da movimentação `MovimentacaoComponent` , conforme o trecho a seguir:

```
declarations: [RequisicaoComponent, MovimentacaoComponent]
```

Dessa forma, finalizamos a integração dos componentes. Ao executar `ng serve -o` e clicar no menu para a rota de requisições já temos os dois componentes na tela, conforme imagem do início do

capítulo. Resta o componente para exibição das movimentações, tarefa que concluiremos na próxima seção.

7.4 Lista de Movimentações

O objetivo nesse componente é exibir todas as movimentações de uma requisição, listando somente as opções de edição que foram registradas pelo usuário logado.

No terminal, criamos o módulo e componente com os comandos:

```
ng g m components/admin/movimentacao/lista --routing
ng g c components/admin/movimentacao/lista
```

Solicitamos a criação do componente dentro da estrutura criada para `movimentacao`. Também não vamos definir uma rota, pois utilizaremos a abordagem de composição de componentes *pai-filho.*

Para não estender, é preciso aplicar as mesmas configurações de importação no módulo `lista.module.ts` e vínculo do componente em `lista-routing.module.ts`, detalhadas nas seções anteriores, substituindo pelas referências atuais.

Na classe do componente `lista.component.ts` temos as seguintes declarações:

```
export class ListaComponent implements OnInit {

  @Input() movimentacoes: Movimentacao[];
  @Input() requisicaoSelecionada: Requisicao;
  @Input() displayDialogMovimentacoes: boolean;
  @Input() funcionarioLogado: Funcionario;
  @Output() displayChange = new EventEmitter();

  listaStatus: string[];
  displayDialogMovimentacao: boolean;
  form: FormGroup;
```

```
  edit: boolean;
  indexMovimentacoes: number;
```

Temos vários objetos cujos valores receberemos de outros componentes, indicados com o decorator `@Input()`, como a lista de movimentação em `movimentacoes`, a requisição selecionada no objeto `requisicaoSelecionada`, o *dialog* que exibe a lista em `displayDialogMovimentacoes` e `funcionarioLogado`, que representa o funcionário logado na aplicação.

Em seguida, temos o decorator `@Output()`, que significa a ordem inversa da emissão do evento, ou seja, enquanto o `@Input()` representa o fluxo do pai para o filho, em `@Output()` o fluxo está na direção do componente filho que emite um evento para o pai.

Na propriedade `displayChange` vamos informar ao componente pai o fechamento do *dialog*. Para isso precisamos instanciar a classe `EventEmitter`.

Vamos precisar de uma variável para guardar o índice da tabela de movimentações. Fazemos isso com `indexMovimentacoes` para realizar as operações de atualização e exclusão.

Continuando o código da classe, temos no método construtor a injeção necessária do serviço e do formulário em:

```
constructor(
  private requisicaoService: RequisicaoService,
  private fb: FormBuilder
) { }
```

Observe que tivemos uma redução considerável de declarações no método construtor por conta do uso de `@Input`, cujos valores serão recebidos da classe pai. No método `ngOnInit()` chamamos dois métodos:

```
ngOnInit() {
  this.configForm();
  this.carregaStatus();
}
```

Esses métodos são responsáveis pela configuração do formulário e a lista de status, conforme implementação a seguir:

```
configForm() {
   this.form = this.fb.group({
     funcionario: new FormControl('', Validators.required),
     dataHora: new FormControl(''),
     status: new FormControl('', Validators.required),
     descricao: new FormControl('', Validators.required)
   })
 }
```

Em `configForm()` temos a definição dos campos no formulário `funcionario`, `dataHora`, `status`, e `descricao` com as validações `Validators.required`.

E o método para exibir as possibilidades de status para um movimento em:

```
carregaStatus() {
   this.listaStatus = ['Aberto', 'Pendente', 'Processando', 'Não
autorizada', 'Finalizado'];
 }
```

A seguir, vamos implementar um método para selecionar a movimentação em:

```
 selecionaMovimento(mov: Movimentacao, index: number) {
   this.edit = true;
   this.displayDialogMovimentacao = true;
   this.form.setValue(mov);
   this.indexMovimentacoes = index;
 }
```

Informamos no parâmetro o movimento e o índice do array com `mov: Movimentacao, index: number`. Definimos o modo de edição e o *dialog* para `true`, além de setar o valor no formulário com `this.form.setValue(mov)`. Concluímos guardando o valor da variável do índice para o atributo `indexMovimentacoes`.

No método para fechar o *dialog* temos:

```
onClose() {
   this.displayChange.emit(false);
 }
```

Basicamente informamos o valor booleano `false` no parâmetro do método em `emit(false)`. Assim conseguimos informar a emissão do evento ao componente pai, fechando o *dialog*.

No método a seguir, vamos implementar o código que atualiza a movimentação:

```
 update() {
   this.movimentacoes[this.indexMovimentacoes] = this.form.value
   this.requisicaoSelecionada.movimentacoes = this.movimentacoes;
   this.requisicaoSelecionada.status = this.form.controls['status'].value
   this.requisicaoSelecionada.ultimaAtualizacao = new Date();
   this.requisicaoService.createOrUpdate(this.requisicaoSelecionada)
     .then(() => {
       this.displayDialogMovimentacao = false;
       Swal.fire(`Movimentação ${!this.edit ? 'salvo' : 'atualizado'} com
sucesso.`, '', 'success');
     })
     .catch((erro) => {
       this.displayDialogMovimentacao = true;
       Swal.fire(`Erro ao ${!this.edit ? 'salvo' : 'atualizado'} o
Movimentação.`, `Detalhes: ${erro}`, 'error');
     })
   this.form.reset()
 }
```

Inicialmente, atualizamos o elemento `this.movimentacoes` com os valores do formulário, com base na posição do array, através do trecho `this.movimentacoes[this.indexMovimentacoes] = this.form.value`.

Atualizamos o array de movimentações em `this.requisicaoSelecionada.movimentacoes = this.movimentacoes;` e o status atual da requisição selecionada com o objeto `this.requisicaoSelecionada.status`.

Além disso, atualizamos o atributo da última atualização com `new Date()`. Na sequência, invocamos a função `createOrUpdate` para efetivar a atualização do registro.

Para realizar a exclusão, inicialmente desenvolvemos uma função auxiliar:

```
remove(array, element) {
   return array.filter(el => el !== element);
 }
```

Há várias abordagens para remover um elemento de um array. Entre elas podemos utilizar o método `filter`. Assim temos o retorno de um novo array com todos os elementos, exceto o informado no parâmetro da função em `el !== element`.

Assim, partimos para a função de exclusão:

```
delete(mov: Movimentacao) {
   const movs = this.remove(this.movimentacoes, mov)
   Swal.fire({
     title: 'Confirma a exclusão da Movimentação?',
     text: "",
     type: 'warning',
     showCancelButton: true,
     confirmButtonText: 'Sim',
     cancelButtonText: 'Não'
   }).then((result) => {
     if (result.value) {
       this.requisicaoSelecionada.movimentacoes = movs;
       this.requisicaoService.createOrUpdate(this.requisicaoSelecionada)
         .then(() => {
           Swal.fire('Movimentação excluída com sucesso!', '', 'success')
           this.movimentacoes = movs;
         })
     }
   })
 }
```

Passamos no parâmetro da função `delete` o objeto de movimento em `mov: Movimentacao`. Definimos uma constante que recebe o array sem o elemento no parâmetro através do trecho: `const movs = this.remove(this.movimentacoes, mov)`.

Observe que não chamamos o método de exclusão no serviço. Diante da confirmação do usuário, em `if (result.value)`, nós atualizamos o registro da requisição com o novo array filtrado nas linhas:

```
this.requisicaoSelecionada.movimentacoes = movs;
this.requisicaoService.createOrUpdate(this.requisicaoSelecionada)
```

A visualização do código completo está em https://bit.ly/2VBK3ps.

Partimos para codificação do template da lista em `lista.component.html`.

Template Lista

Vamos abrir o arquivo em `lista.component.html`. O componente será exibido no `dialog`, dessa forma, iniciamos a codificação em:

```
<p-dialog header="Movimentações" [minWidth]="600" [contentStyle]="
{'overflow':'visible'}"
 [(visible)]="displayDialogMovimentacoes" (onHide)="onClose()"
[responsive]="true" [modal]="true">
```

Definimos as propriedades do *dialog* e o método `(onHide)="onClose()"`, que definimos na lógica do componente anteriormente como `@Output`.

Na sequência, fazemos o HTML da tabela para exibir os movimentos em:

```
<table class="table table-striped table-hover table-bordered col-
centered">
     <thead class="thead-dark">
       <tr>
         <th class="text-center">Data</th>
```

```
          <th class="text-center">Funcionario</th>
          <th class="text-center">Status</th>
          <th class="text-center">Ações</th>
        </tr>
      </thead>
      <tbody>
        <tr *ngFor="let movimento of movimentacoes; index as i">
          <td class="text-center">{{movimento.dataHora.seconds * 1000 |
date : 'dd/MM/yyyy HH:mm'}}</td>
          <td class="text-center">{{movimento.funcionario.nome}}</td>
          <td class="text-center">{{movimento.status}}</td>
```

Definimos o cabeçalho e as propriedades de exibição para `Data` , `Funcionario` e `Status` .

Na iteração do array , utilizamos novamente o operador `*ngFor` . Observe que criamos uma variável para guardar o índice com `index as i` . Novamente, utilizamos um *pipe* para formatar a data em `| date : 'dd/MM/yyyy HH:mm'` .

Nas células dos botões, codificamos da seguinte maneira:

```
<td class="text-center">
            <div *ngIf="funcionarioLogado.email ===
movimento.funcionario.email">
              <button type="button"
(click)="selecionaMovimento(movimento,i)" class="btn btn-success ">
                <i class="fas fa-edit"></i>
              </button>
              <button type="button" (click)="delete(movimento)" class="btn
btn-danger">
                <i class="fas fa-trash"></i>
              </button>
            </div>
          </td>
```

Condicionamos a exibição das colunas em um `div` com base no email do funcionário logado e do registro de quem criou a movimentação em `*ngIf="funcionarioLogado.email === movimento.funcionario.email"` .

O resultado da condição está na imagem a seguir.

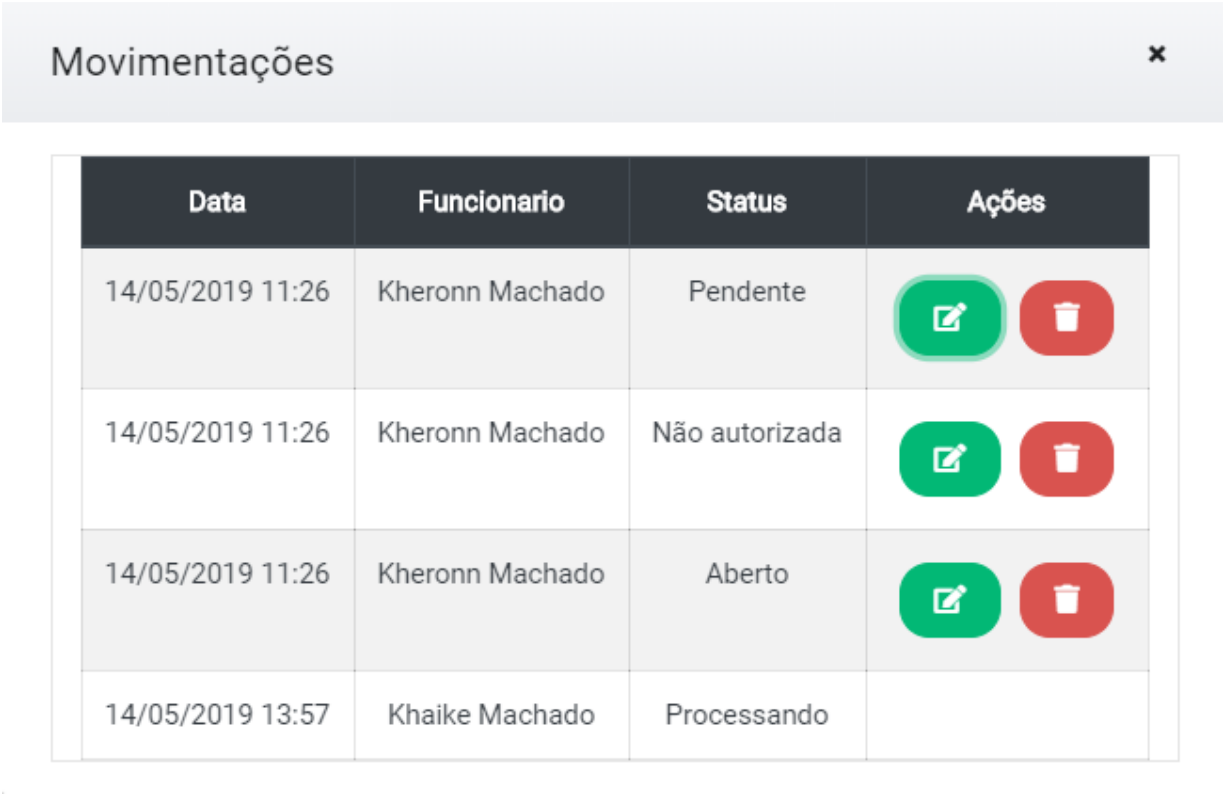


Figura 7.5: Movimentações

Observe que o último registro não exibe os botões na coluna de ações pois foram lançados por um usuário diferente do funcionário logado.

No método `selecionaMovimento(movimento,i)` informamos o movimento e o índice no parâmetro.

Na sequência, temos o *dialog* exatamente igual ao que fizemos no componente anterior. Poderia ser um componente reutilizável, não? Fica o exercício.

O código completo do template está disponível em https://bit.ly/2EcwMsD.

Novamente, para que exiba o componente no navegador, devemos informar a *tag* desse componente no componente pai.

Insira no final do arquivo em `movimentacao.component.html` .

```
<app-lista [displayDialogMovimentacoes]="displayDialogMovimentacoes"
[movimentacoes]="movimentacoes"
 [requisicaoSelecionada]="requisicaoSelecionada"
[funcionarioLogado]="funcionarioLogado"
(displayChange)="onDialogClose($event)"></app-lista>
```

Vinculamos os valores da classe pai em `movimentacao.component.ts` aos atributos de entrada `[displayDialogMovimentacoes]` , `[movimentacoes]` , `[requisicaoSelecionada]` e ao de saída em `(displayChange)` da classe filho `lista.component.ts` .

A imagem a seguir apresenta a composição dos componentes.

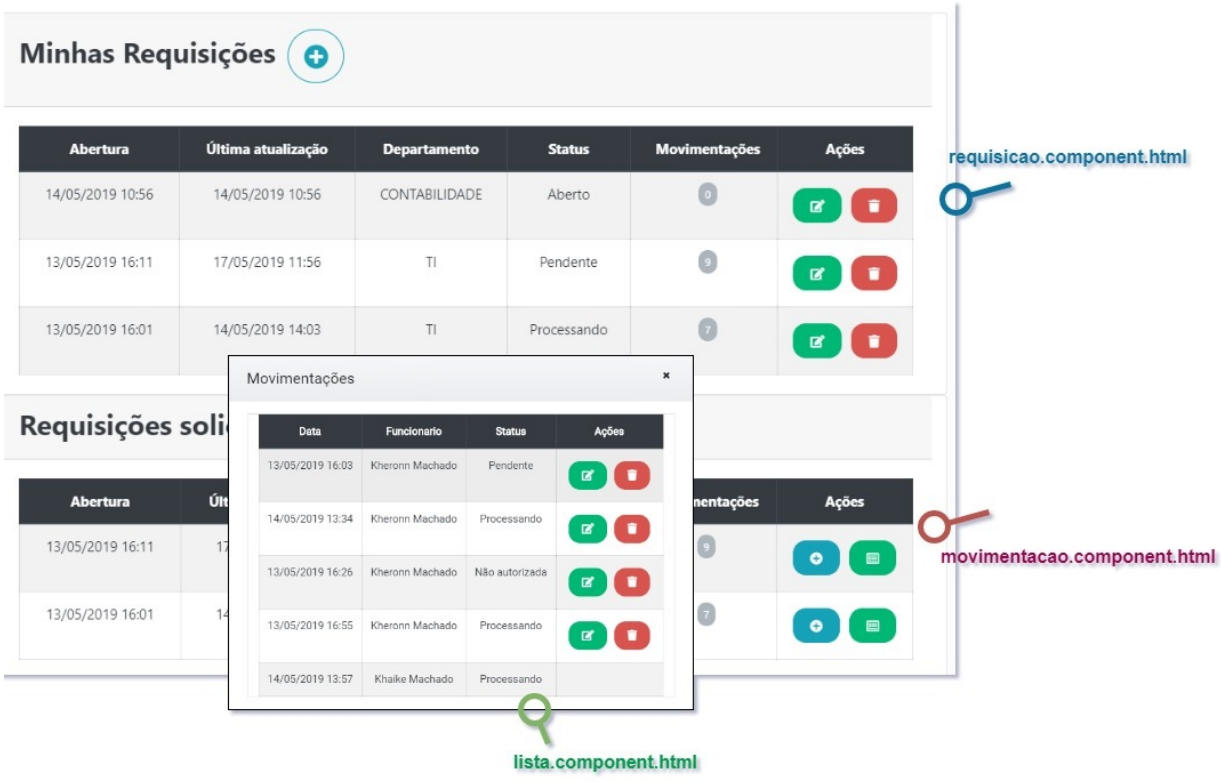


Figura 7.6: Composição dos componentes

Para finalizar, declaramos o componente no array em `declarations` , incluindo o componente `ListaComponent` em `requisicao.module.ts` , conforme instrução a seguir:

```
  declarations: [RequisicaoComponent, MovimentacaoComponent,
ListaComponent],
```

Por enquanto, conforme aumentam os lançamentos das requisições, o conteúdo da página se estende, e perdemos a posição do *scroll* (barra de rolagem) ao navegar entre os componentes.

Para corrigir esse comportamento, devemos editar o arquivo de rotas da aplicação em `app-routing.module.ts` na pasta `app`.

No array de `imports` insira na frente de `routes` a opção `scrollPositionRestoration`:

```
RouterModule.forRoot(routes, {scrollPositionRestoration: 'enabled'})]
```

A opção `enabled` restaura as posições de rolagem durante a navegação. Ao navegar para a frente, a posição de rolagem será definida para as coordenadas `[0, 0]`.

E assim finalizamos a implementação do requisito *Gerenciar Requisições*.

Nos próximos capítulos vamos explorar outros recursos do Firebase na implementação dos requisitos da aplicação.

Capítulo 8
Firebase Cloud Storage - Salvando arquivos estáticos

Além dos serviços de banco de dados e autenticação já explorados na construção da nossa aplicação, podemos utilizar outro serviço da Firebase, o **Cloud Storage**.

O Cloud Storage permite o armazenamento de arquivos, como imagens, vídeos ou outros conteúdos do usuário.

Assim como outros recursos da plataforma do Firebase, esse serviço também é escalável e seguro, permitindo a criação de regras de segurança nas operações de download e upload.

O objetivo deste capítulo é a integração do serviço na nossa aplicação, através da atualização de uma funcionalidade no componente de funcionários.

Vamos implementar a possibilidade de realizar o upload de foto do funcionário.

8.1 Configurando as regras de acesso

Antes de implementar o código, vamos acessar o console do Firebase (https://console.firebase.google.com) e clicar no menu `Storage`.

Na primeira vez em que acessamos, temos o seguinte resultado:

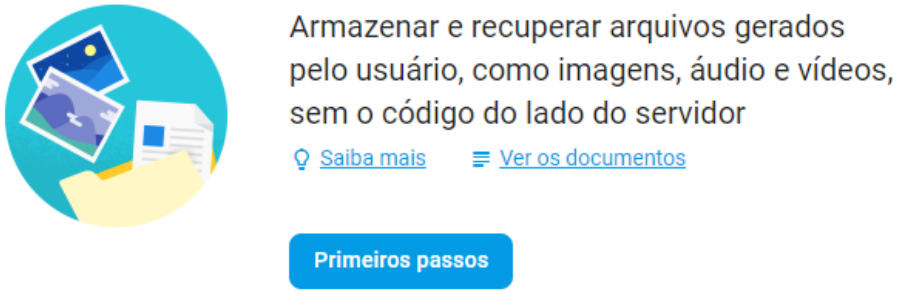

Armazenar e recuperar arquivos gerados pelo usuário, como imagens, áudio e vídeos, sem o código do lado do servidor
Saiba mais
Ver os documentos
Primeiros passos

Figura 8.1: Home do Serviço de Storage

Há um link explicando o serviço e a documentação dos recursos. Vamos clicar no botão `Primeiros passos`.

Na sequência, é mostrado um *dialog* para a definição das regras de segurança.

Regras de segurança
Por padrão, suas regras permitem todas as leituras e gravações de usuários autenticados.
Depois de definir sua estrutura de dados, será necessário criar regras para proteger seus dados.
Saiba mais
service firebase.storage {
match /b/{bucket}/o {
match /{allPaths=**} {
allow read, write: if request.auth != null;
}
}
}
Cancelar
Ok

Figura 8.2: Regras de Segurança

O padrão é a permissão de todas operações de acesso e gravação de usuários autenticados.

Essa regra já atende aos nossos requisitos, pois queremos que o usuário possa realizar o *upload* de fotos somente após o login. Confirmamos no botão *Ok*.

Após a confirmação, uma tela com uma tabela vazia é apresentada. Esse é o local de acesso aos arquivos e pastas que o usuário salvar.

Ao clicar na segunda guia temos a regra atual:

```
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
```

```
      allow read, write: if request.auth != null;
    }
  }
}
```

Essas definições são utilizadas para definir quem tem acesso à leitura e gravação dos arquivos, representadas respectivamente por `read` e `write`.

As regras devem especificar primeiro o tipo de serviço, nesse caso o `firebase.storage`, e o intervalo do Cloud Storage com `match /b/{bucket}/o`. Além disso elas estão associadas aos caminhos dos recursos com `match`

O tipo básico de regra é o `allow`. Após as operações, temos a opção de definir uma condição precedida de dois pontos, com `if request.auth != null`. Nesse caso, a avaliação da solicitação verifica na variável `request` o usuário autenticado em `auth`.

Já podemos retornar ao projeto e codificar uma função que utilize o serviço.

8.2 Lógica e template para upload de fotos do funcionário

Na classe do componente do funcionário `funcionario.component.ts` vamos incluir alguns objetos para realizar a função.

Abaixo dos objetos já declarados inclua o código:

```
@ViewChild('inputFile', { static: true }) inputFile: ElementRef;
uploadPercent: Observable<number>;
downloadURL: Observable<string>;
task: AngularFireUploadTask;
complete: boolean;
```

No primeiro objeto utilizamos um decorator do tipo `@ViewChild` para acessar o componente. O nome informado nos parênteses, `'inputFile'`, deve ter uma referência no template, que faremos posteriormente. Além disso, incluímos o atributo `static`, que se encarrega de verificar os resultados da consulta antes da execução da detecção de alteração no componente.

Em `uploadPercent`, escrevemos `Observable<number>` para acompanhar o progresso do upload. Já a variável `downloadURL` é utilizada para capturar o endereço do recurso no servidor, e o objeto `task` do tipo `AngularFireUploadTask` é uma interface para tarefas no storage.

O atributo `complete` será responsável para marcar o início e fim do processo.

Vamos precisar da classe `AngularFireStorage` para realizar as operações. Incluímos um objeto, injetando no construtor:

```
constructor(
   private storage: AngularFireStorage,
```

Dessa forma, já podemos escrever a função para o envio de arquivos:

```
async upload(event) {
   this.complete = false;
   const file = event.target.files[0];
   const path = `funcionarios/${new Date().getTime().toString()}`;
   const fileRef = this.storage.ref(path);
   this.task = this.storage.upload(path, file);
   this.task.then(up => {
     fileRef.getDownloadURL().subscribe(url => {
       this.complete = true;
       this.form.patchValue({
         foto: url
       })
     });
   });
   this.uploadPercent = this.task.percentageChanges();
```

```
    this.inputFile.nativeElement.value = '';
  }
```

Codificamos o método `upload(event)` passando o parâmetro `event` vindo do template. Iniciamos a marcação do processo com `this.complete = false` e recuperamos o arquivo com o objeto `file`.

Definimos um caminho com `path`, criando uma pasta `funcionarios` e nomeando o arquivo dinamicamente com a hora da instância da data através de `new Date().getTime().toString()`.

Informamos a referência do caminho ao `storage` e invocamos o método `upload(path, file)` para efetivar a operação do envio da imagem.

Com o objeto `task`, conseguimos recuperar o caminho, atualizando o campo do formulário no trecho:

```
fileRef.getDownloadURL().subscribe(url => {
      this.complete = true;
      this.form.patchValue({
        foto: url
      })
    });
```

E monitoramos o progresso do envio com `this.uploadPercent = this.task.percentageChanges()`, além de limpar o campo de upload com `this.inputFile.nativeElement.value = ''` para as próximas ações.

Vamos incluir no template do componente uma nova coluna para exibir a imagem e um campo para a seleção da imagem.

Em `funcionario.component.html`, incluímos na seção da tabela uma nova coluna, com:

```
<th class="text-center">Foto</th>
```

E uma nova célula na linha da exibição:

```
  <td class="text-center">
  <img [src]="funcionario.foto || '/assets/imgs/no-image.png'"
```

```
style="width: 100px" class="img-fluid">
 </td>
```

Definimos no caminho da imagem a nova propriedade em `[src]="funcionario.foto`. Ainda dentro do valor, utilizamos as duas barras de pé, `||`. Essa marcação indica que, se não existir um valor para imagem em `funcionario.foto`, deve-se utilizar uma imagem estática no caminho `'/assets/imgs/no-image.png'`.

Após a finalização da implementação, essa será a nova aparência da tabela.

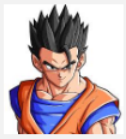
Foto	Nome	Email	Departamento	Função	Ações
	Cael Machado	kheronn@seed.pr.gov.br	CONTABILIDADE	Controlador	
	Kheronn Machado	kheronn@gmail.com	TI	Programador	
	Khaike Machado	kheronn.machado@escola.pr.gov.br	RH	Coordenador	

Figura 8.3: Funcionários com imagens

Para completar a implementação do *upload* no formulário, vamos incluir o seguinte código abaixo do campo de função:

```
<div class="ui-grid-row mb-3">
       <div class="ui-grid-col-4">
         <label for="foto">Foto:</label>
       </div>
       <div class="ui-grid-col-8 ">
        <input type="file" #inputFile class="form-control"
(change)="upload($event)" />
        <progress style="width: 100%;" max="100" [value]="(uploadPercent |
async)"></progress>
```

```
    </div>
  </div>
```

Incluímos o campo do tipo com `type="file"` para escolha de arquivos no computador. Definimos uma variável de template em `#inputFile`, conforme mencionado na classe do componente.

No método `(change)="upload($event)"`, chamamos a função implementada anteriormente para o upload.

Finalizamos a implementação com a declaração de uma barra de progresso com o componente `<progress`, vinculando o valor em `[value]` ao objeto que codificamos na classe `uploadPercent`, sem esquecer de usar o pipe `| async` para operações assíncronas.

O formulário terá o seguinte visual:

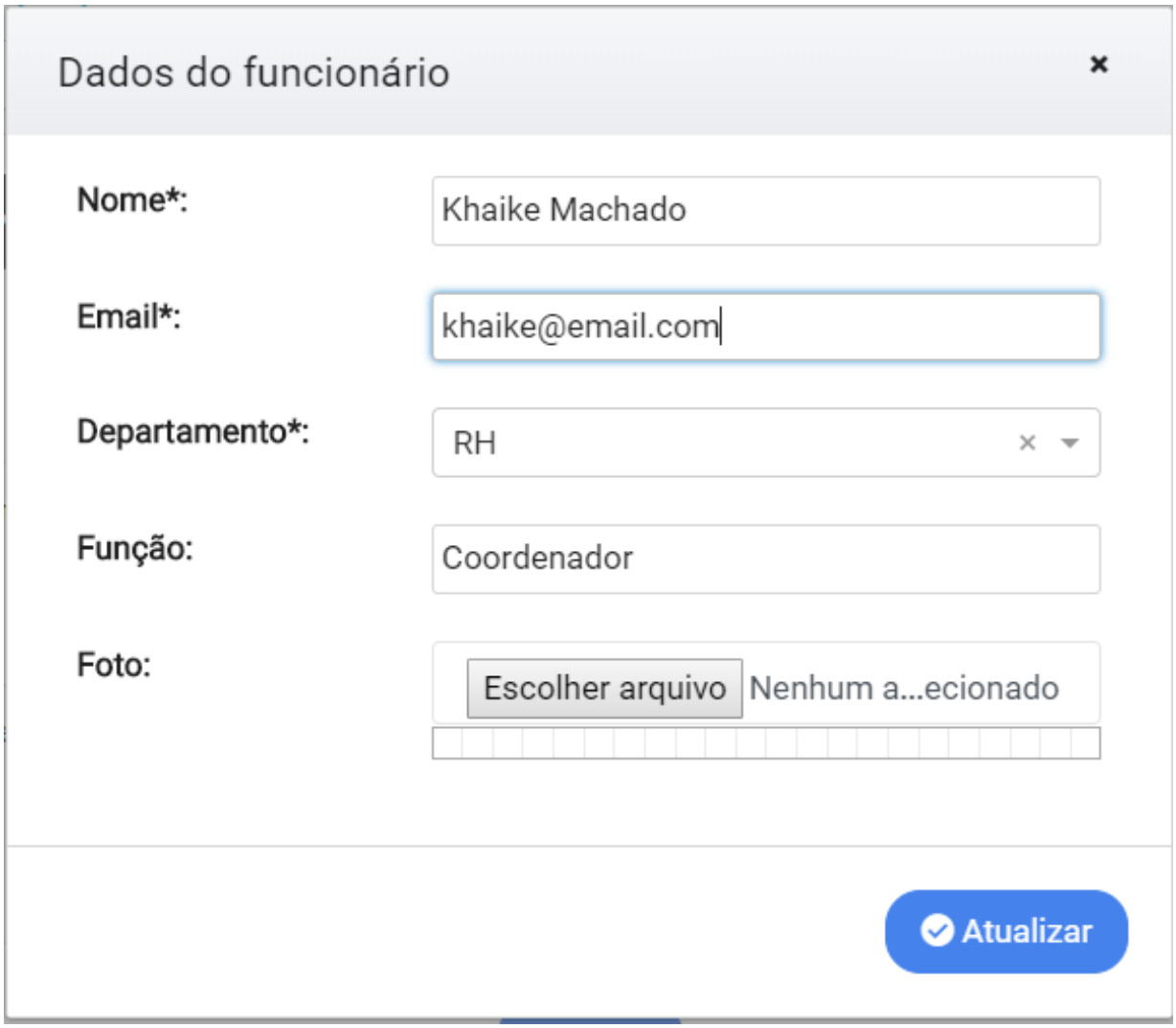


Figura 8.4: Formulário funcionário com o campo para o upload

Assim que incluímos mais registros, acessando o console do Firebase e, clicando no menu Storage, temos a pasta de Funcionários:

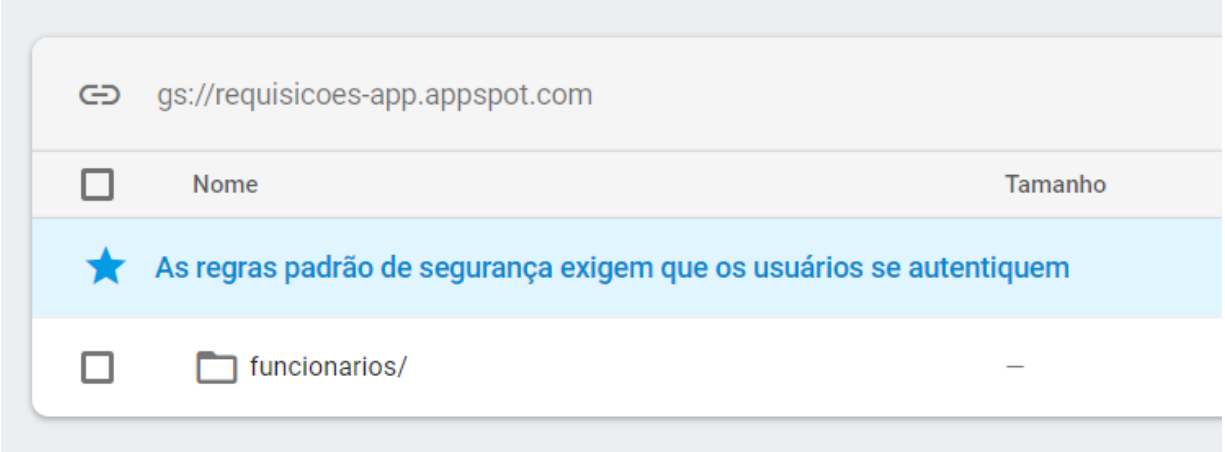


Figura 8.5: Pasta Funcionários

Clicando na pasta, temos acesso aos arquivos armazenados com segurança:

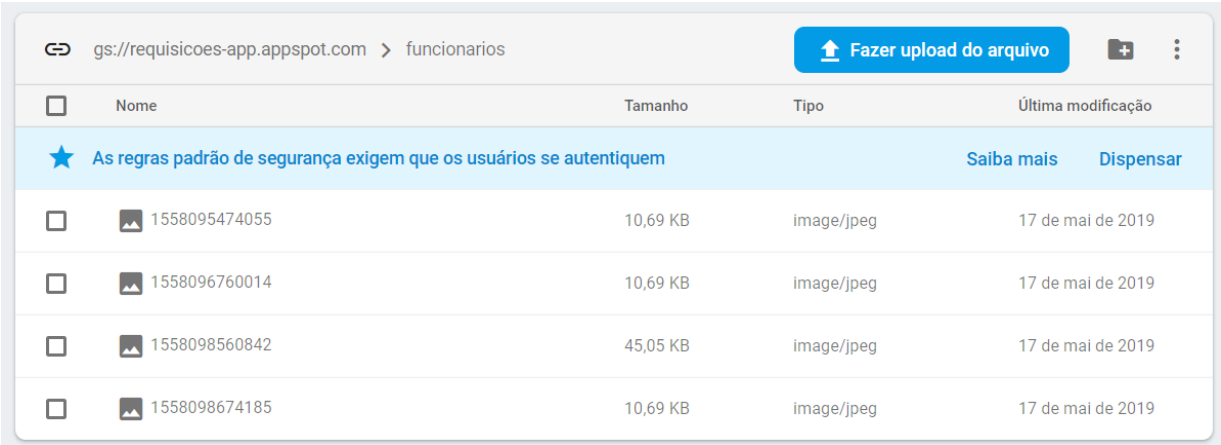


Figura 8.6: Arquivos no Storage

Assim concluímos o capítulo com a utilização de mais um serviço da plataforma Firebase.

A partir dessa implementação é possível ampliar o uso dos recursos de forma fácil e personalizar conforme a necessidade.

No próximo capítulo vamos explorar o desenvolvimento de funções executadas no lado do servidor.

Capítulo 9
Firebase Cloud Functions - Criação de usuário e envio de emails

Uma demanda comum no desenvolvimento de aplicações está relacionada à execução de operações no servidor. Isso porque há cenários em que a execução do lado do cliente exige muito recurso de memória ou uso da banda.

Entre os casos podemos citar operações de manutenção na base de dados, disparo de emails com base em eventos, notificações e outros.

Assim, ter a disposição um ambiente do lado do servidor que realize essas ações ao invés da aplicação é uma forma adequada no desenvolvimento de sistemas escaláveis e que otimizem recursos.

Com o Cloud Functions para Firebase, conseguimos executar o código de back-end automaticamente em resposta a eventos acionados pelos recursos do Firebase. A plataforma se encarrega de gerenciar e dimensionar automaticamente os recursos computacionais.

Neste capítulo vamos explorar esse recurso na implementação dos requisitos *Criar usuário* e *Notificar usuário*.

Detalhando as operações, no primeiro requisito, vamos invocar uma função no servidor que crie um usuário com email e senha toda vez que o evento de inserção na coleção de funcionários for disparado.

Na segunda operação, queremos que o funcionário que lançou uma requisição seja informado por email toda vez que houver movimentação.

Nas próximas seções vamos configurar nosso projeto e codificar as funções necessárias.

9.1 Firebase CLI

O primeiro passo para utilizar o recurso será a instalação de uma interface de linha de comando, o **Firebase CLI**.

Através dele conseguimos uma série de ferramentas para gerenciar, visualizar e implantar projetos do Firebase.

No terminal, informe:

```
npm install -g firebase-tools
```

Uma vez instalado, devemos realizar o login. Ainda no terminal digite:

```
firebase login
```

Utilize as mesmas credenciais utilizadas no capítulo *3. Firebase*. Dessa forma, conectamos a máquina local ao Firebase, com acesso aos nossos projetos.

No próximo comando, o terminal solicitará através de etapas, algumas configurações para criação do projeto relacionado às funções.

Você pode criar uma pasta específica para ou projeto ou utilizar a estrutura da aplicação em Angular. Eu utilizarei a última por razões de versionamento de código, assim, com o terminal na pasta do projeto, informe:

```
firebase init functions
```

O comando inicia um diretório `functions` na raiz do projeto, solicitando qual o projeto no Firebase vamos vincular. No caso, escolhi `requisicoes-app` , conforme ilustração a seguir:

```
=== Project Setup

First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

? Select a default Firebase project for this directory:
  [don't setup a default project]
  devdata-web (WebSite)
> requisicoes-app (requisicoes-app)
  [create a new project]
```

Figura 9.1: Vinculando o projeto

A próxima pergunta informa que será criado um projeto Node.js com os pacotes pré-configurados e que as funções podem ser implantadas, e solicita a linguagem que utilizaremos para escrever o código. Vamos usar o JavaScript:

```
=== Functions Setup

A functions directory will be created in your project with a Node.js
package pre-configured. Functions can be deployed with firebase deploy.

? What language would you like to use to write Cloud Functions? (Use arrow keys)
> JavaScript
  TypeScript
```

Figura 9.2: Selecionando a linguagem

O último passo questiona se desejamos instalar as dependências. Confirmamos com `Y`.

```
+  Wrote functions/package.json
+  Wrote functions/index.js
+  Wrote functions/.gitignore
? Do you want to install dependencies with npm now? (Y/n)
```

Figura 9.3: Instalando as dependências

Depois de um momento já temos a pasta `functions` com os arquivos das dependências e o `index.js`, onde codificaremos as funções.

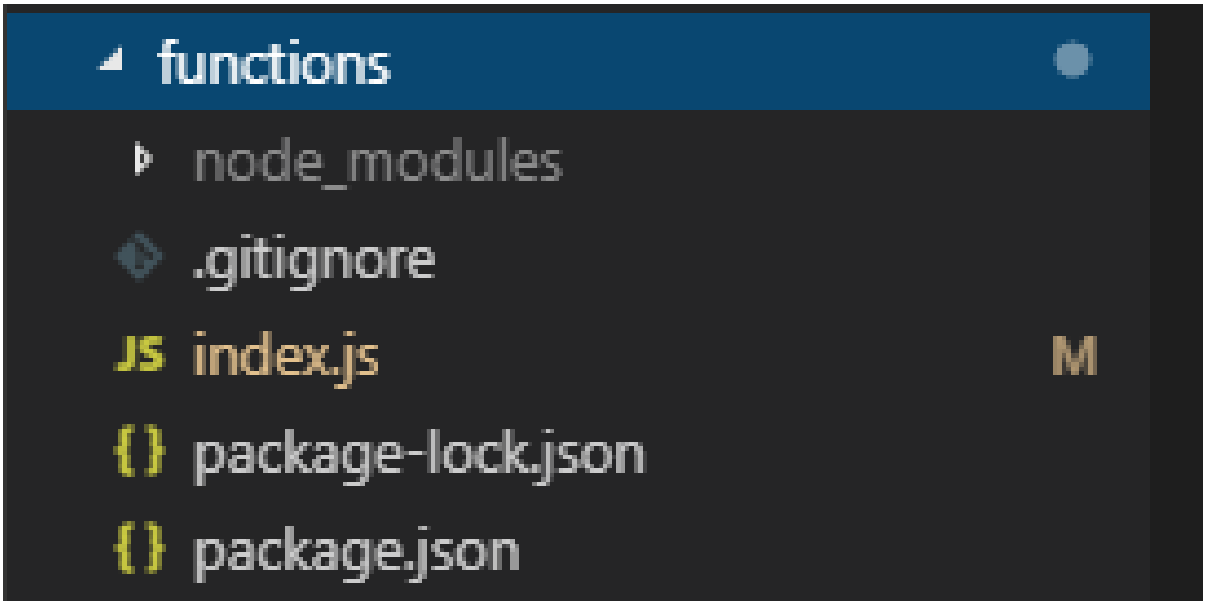


Figura 9.4: Conteúdo da pasta functions

Na raiz do projeto também foram criados dois novos arquivos:

```
* firebase.jon  Arquivo que lista as configurações do projeto;
* .firebaserc   Armazena os aliases de projeto.
```

Para enviar e-mails vamos precisar de uma instalação adicional. Navegue até a pasta do projeto:

```
cd functions
```

E na sequência, digite:

```
npm install nodemailer --save
```

Assim, instalamos o *Nodemailer*, que é um módulo para aplicações Node.js que permite o envio de e-mails.

Dessa forma já temos realizado o setup inicial para utilização do Cloud Functions.

Na próxima seção vamos criar a primeira função que cria um usuário para logar na aplicação.

9.2 Função para criar um usuário

Vamos abrir o arquivo `functions/index.js` e começar a codificação. Começamos definindo os *imports* necessários em:

```
const functions = require('firebase-functions');
const admin = require('firebase-admin');
const nodemailer = require('nodemailer');
admin.initializeApp();
```

Inicialmente precisamos importar os módulos do Cloud Functions e do SDK Admin usando instruções `require` do Node.

Em `admin` , temos por exemplo acesso aos seguintes recursos:

- Ler e gravar dados no Realtime DataBase;
- Gerar e verificar tokens de autenticação;
- Acesso ao Cloud Firetore.

Em `nodemailer` carregamos o módulo para envio de emails e, na linha `admin.initializeApp()` , inicializamos uma instância de `admin` .

O próximo passo é definir as credenciais de uma conta que utilizaremos para envio de emails.

```
let mailTransport = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: sem-email@gmail.com',
    pass: sua-senha'
  }
});
```

Definimos uma variável `mailTransport` para criar um transporte, passando o tipo do serviço e as credenciais para autenticação. Informe seu e-mail e senha.

Os dois passos a seguir são de extrema importância para habilitar o Gmail no envio de emails através de aplicações de terceiros:

1. Habilitar no endereço https://bit.ly/124TgWN o acesso a apps menos seguras.
2. Permitir acesso à sua conta Google em https://bit.ly/2HETuL6.

Sobre habilitar "apps menos seguras", o Google define assim para enfatizar os riscos de integrar com APIs de terceiros.

Sem esses procedimentos **NÃO É POSSÍVEL ENVIAR EMAILS COM GMAIL** pois você terá problemas relacionados à permissão.

Seguindo o código, vamos escrever a função que "escuta" um evento no banco de dados e capturar os dados:

```
exports.createUser = functions.firestore
 .document('/funcionarios/{documentId}')
 .onCreate((snap, context) => {

   const funcionario = snap.data();
   const email = funcionario.email;
   const nome = funcionario.nome;
```

Definimos uma função nomeando como `createUser`. Através do objeto `firestore` temos acesso aos eventos do banco. Na sequência, acessamos a coleção `/funcionarios` utilizando um coringa `{documentId}`. Dessa forma, conseguimos ativar o evento `onCreate` para qualquer documento.

Continuando a explicação do código, definimos três variáveis `funcionario`, `email` e `nome` que receberão no *callback* do evento o documento criado através do objeto `snap`.

Para entender o método, há possibilidade de gerenciar 4 tipos de eventos:

Tipo	Acionador
`onCreate`	Quando um documento é gravado pela primeira vez
`onUpdate`	Quando um documento já existe e tem algum valor alterado
`onDelete`	Quando um documento é excluído
`onWrite`	Acionado quando qualquer outros três eventos forem adicionados

Na sequência recuperamos o valor da inserção em `snap.data()` e o atribuímos à variável `funcionario`.

Através dessa propriedade conseguimos recuperar as informações do funcionário, como o e-mail, por exemplo.

Na sequência temos o trecho que invocamos para criar o usuário do Firebase:

```
return admin.auth().createUser({
     uid: `${email}`,
     email: `${email}`,
     emailVerified: false,
     password: `123456`,
     displayName: `${nome}`,
     disabled: false
   }).then((userRecord) => {
     console.log('Usuário registrado com sucesso')
     return userRecord;
   })
     .catch(function (error) {
       console.log("Não foi possível criar o usuário:", error);
     });
   })
```

Passamos as informações do funcionário para o método `admin.auth().createUser` , como um identificador único `uid` , o `email` , a senha `password` .

Estamos definindo a senha para todos os usuários criados como `123456` . Aqui você pode definir uma regra de negócio própria.

Por fim, finalizamos com a exibição no `console.log` para a *Promise* concluída `then` ou exibindo o erro com `cacth` .

Podemos testar a função, realizando a implantação no Firebase. No terminal, informe:

```
firebase deploy
```

Se tudo estiver correto, no terminal aparecerá a mensagem de que foi implantado com sucesso. Acesse o https://console.firebase.google.com e clique no menu `Functions` .

No painel central aparece o nome da função implantada:

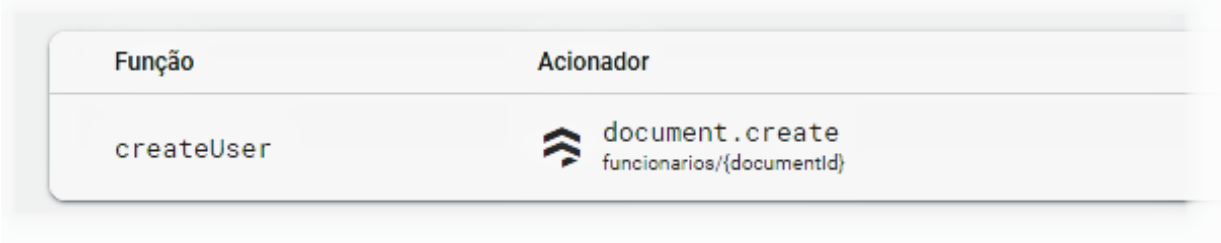


Figura 9.5: Função createUser()

Para testar a função devemos subir a aplicação (`ng serve -o`) e realizar um novo registro de funcionário.

Ao retornar no console do Firebase, menu `Functions` , clique na guia `Registros` .

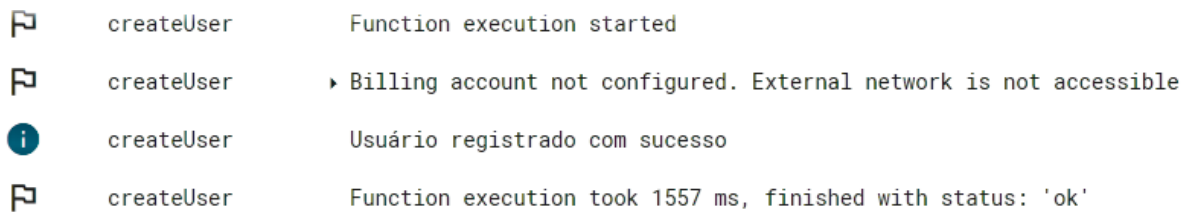

createUser Function execution started
createUser Billing account not configured. External network is not accessible
createUser Usuário registrado com sucesso
createUser Function execution took 1557 ms, finished with status: 'ok'

Figura 9.6: Função createUser()

Temos o resultado da execução da função com a exibição das mensagens. A mensagem que se inicia com `Billing account not configured` significa que estamos utilizando um plano do Firebase, o *Spark*, com restrições e limites de cotas. Para uso comercial e dependendo da quantidade de acessos, o desenvolvedor deve escolher outro plano.

Portanto, o funcionário já tem acesso ao sistema de requisições com o e-mail informado no seu cadastro.

9.3 Função para notificar um usuário - Enviar e-mails

Ainda na codificação do arquivo `index.js` , vamos criar uma função para disparar um e-mail toda vez que uma requisição receber uma atualização.

Na seção anterior, já tínhamos definido um objeto que utiliza a biblioteca *Nodemailer* para enviar e-mails. Para mais detalhes da biblioteca, acesse: https://nodemailer.com/about.

Vamos analisar o código da função `notifyUser` que começa assim:

```
exports.notifyUser = functions.firestore
 .document('/requisicoes/{documentId}')
```

```
.onUpdate((snap, context) => {

  const requisicao = snap.after.data();
  const solicitante = requisicao.solicitante;
  const email = solicitante.email;
```

Novamente, vamos utilizar um evento do *firestore*, mas desta vez será da coleção `requisicoes`. O evento em questão será ativado quando um documento da requisição sofrer atualização. Para recuperar os valores, devemos usar a propriedade `after.data()`.

Atribuímos os valores a uma variável que será a `requisicao`. Depois disso, recuperamos o `solicitante` e o `email`.

Continuando, temos:

```
const movimentacoes = requisicao.movimentacoes;

  if (movimentacoes.length > 0) {
    const movimentacao = movimentacoes[movimentacoes.length - 1];

    const texto = `<h2> Sua requisição recebeu uma atualização! </h2>
                   <h3> Descricao:   ${movimentacao.descricao} </h3>
                   <h4> Status:  ${movimentacao.status}  <br> `
    const mailOptions = {
       from: `<noreply@firebase.com>`,
       to: email,
       subject : `Sistema de Requisições | Processamento de Requisições`,
       html : `${texto}`
     };
    return mailTransport.sendMail(mailOptions).then(() => {
      console.log('Email enviado para:', email);
      return null;
    }).catch((error) => {
      console.log("Não foi possível notificar  o usuário:", error);
    });
  }
})
```

Recuperamos o array de `movimentacoes` e verificamos o tamanho em `if (movimentacoes.length > 0)`.

No caso de existir pelo menos uma movimentação, escrevemos a lógica, recuperando a última movimentação do array em `movimentacoes[movimentacoes.length - 1]`.

Na sequência, vamos desenvolver um texto para o corpo do email, utilizando tags HTML para formatação:

```
const texto = `<h2> Sua requisição recebeu uma atualização! </h2>
                <h3> Descricao:   ${movimentacao.descricao} </h3>
                <h4> Status:  ${movimentacao.status}  <br> `
```

Com a variável `movimentacao`, temos acesso aos atributos para exibição como a `${movimentacao.descricao}` e o `${movimentacao.status}`.

Definimos um objeto para opções do email em `mailOptions`. Aqui declaramos o destino em `to`, o título da mensagem em `subject` e o corpo do email, em `html`.

Por fim, invocamos o método para enviar passando o objeto com as opções em `mailTransport.sendMail(mailOptions)`.

E assim finalizamos a escrita do método. Para ver o código na íntegra acesse: https://bit.ly/2HHXdat.

Para realizar o teste dessa função você deve:

1. Criar uma requisição com seu usuário com email verdadeiro;
2. Criar uma movimentação para essa requisição (qualquer usuário);
3. Verificar na caixa de entrada a mensagem.

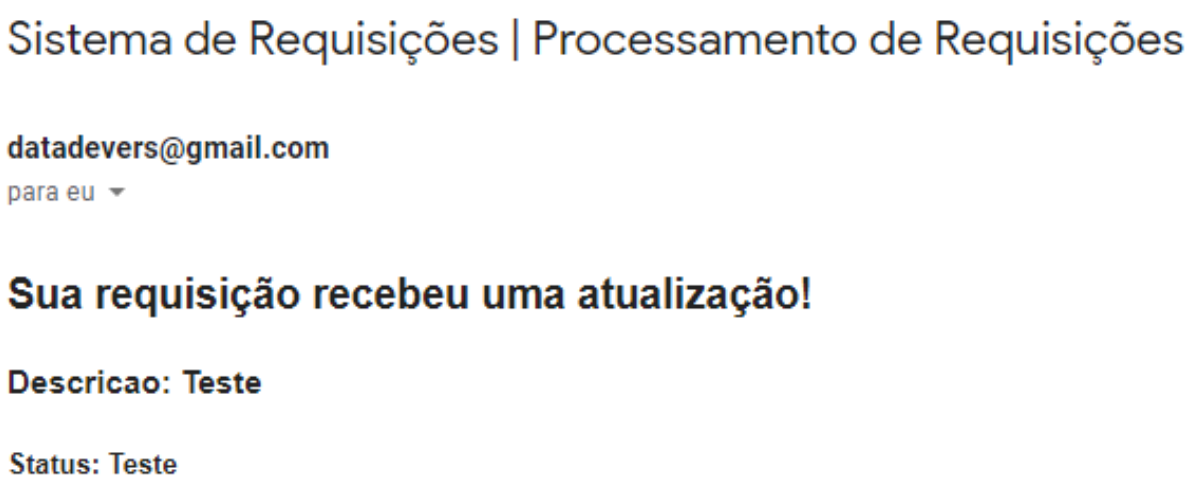


Figura 9.7: Email de notificação

Dessa forma, conseguimos realizar a integração da aplicação com mais um recurso do Firebase para execução de códigos no lado do servidor.

No próximo e último capítulo vamos implantar a aplicação desenvolvida utilizando mais um serviço do Firebase.

Capítulo 10
Deploy da aplicação e considerações finais

Neste último capítulo, vamos fazer o deploy da aplicação utilizando o serviço Firebase Hosting.

Esse recurso oferece hospedagem rápida e segura, além de conseguirmos implantar o projeto com um único comando.

10.1 Firebase Hosting

No capítulo anterior, instalamos o *firebase-tools*, ferramenta necessária para implantação no servidor. Assim, o que faremos a seguir é criar o projeto para produção.

No terminal informe o comando:

```
ng build --prod
```

O parâmetro `build` gera nossa aplicação utilizando a ferramenta Webpack, com opções de configuração padrão especificadas no arquivo de configuração do arquivo `angular.json` .

A opção `--prod` otimiza a construção e o tamanho final da aplicação.

Será gerada uma pasta `dist` no diretório raiz do projeto contendo uma pasta com os arquivos do projeto, prontos para a implantação.

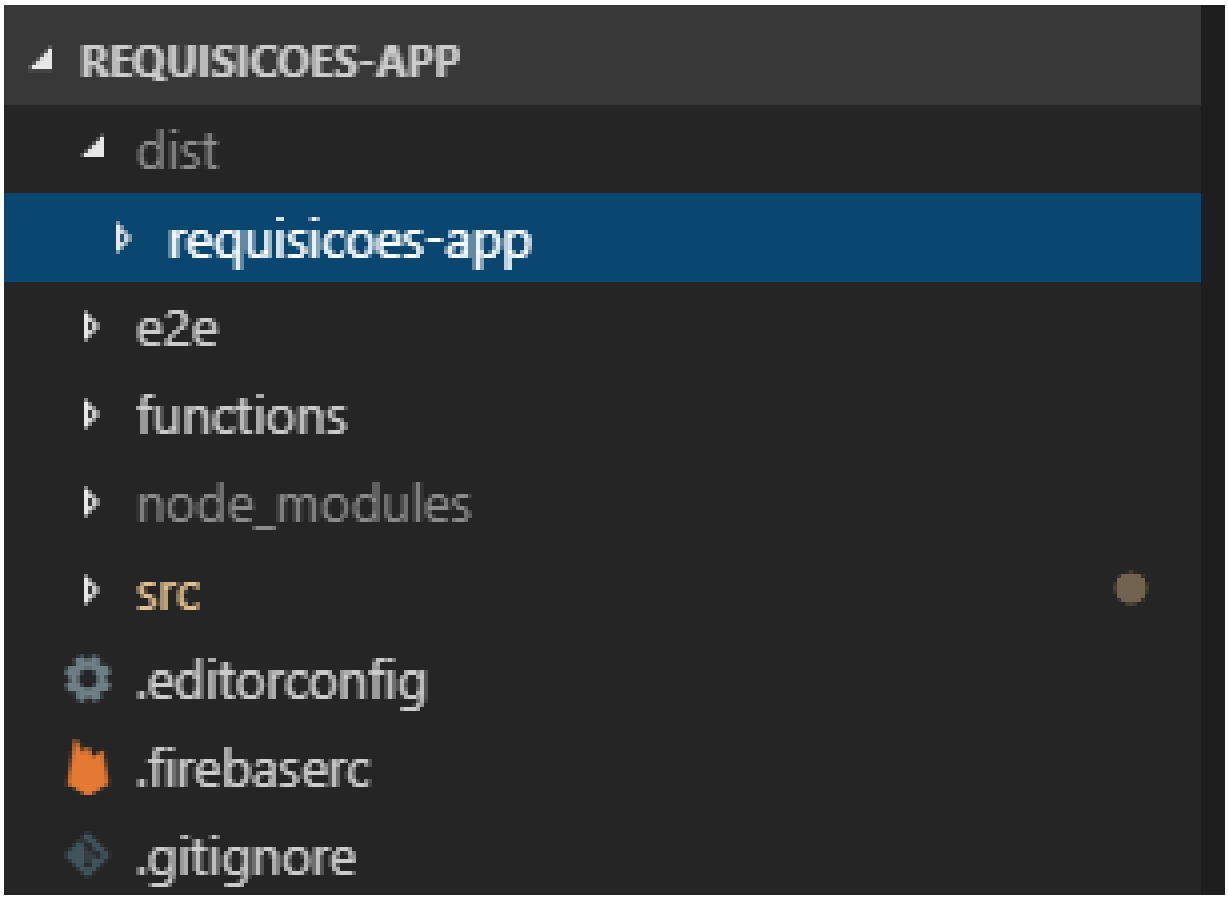


Figura 10.1: Pasta com os projeto para produção

O próximo passo é configurar a hospedagem. No terminal informe:

```
firebase init hosting
```

e na sequência, `Y` para continuar. O comando iniciará as configurações solicitando algumas informações. O terminal exibe a mensagem de que já existe um arquivo `.firebaserc` definido para o projeto, conforme imagem a seguir.

```
Before we get started, keep in mind:

  * You are initializing in an existing Firebase project directory

? Are you ready to proceed? Yes

=== Project Setup

First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

i  .firebaserc already has a default project, skipping

=== Hosting Setup
```

Figura 10.2: Iniciando as configurações de hospedagem

Isso ocorre porque já fizemos no capítulo anterior o mesmo comando `firebase init` para implantar as funções no servidor. Porém, note que dessa vez utilizamos no final do comando a opção `hosting`, para explicitar o tipo de recurso que vamos associar.

A pergunta seguinte solicita o nome do diretório público que vamos enviar para o servidor. Informamos `dist/requisicoes-app`, conforme os passos que fizemos no começo do capítulo ao gerar o projeto para produção.

```
? What do you want to use as your public directory? dist/requisicoes-app
? Configure as a single-page app (rewrite all urls to /index.html)? (y/N) Y
```

Figura 10.3: Configuração de Diretório e SPA

Confirmamos também com `Y` para o tipo da aplicação que é Single Page App (SPA).

A última interação pergunta se queremos sobrescrever o arquivo `index.html`. Informamos com `N`, pois nosso projeto já possui o arquivo.

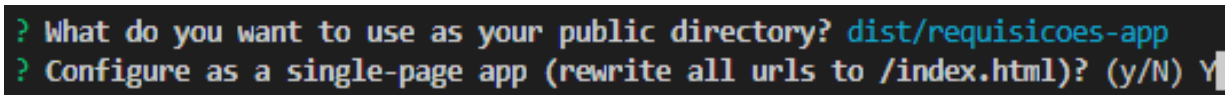

? What do you want to use as your public directory? dist/requisicoes-app
? Configure as a single-page app (rewrite all urls to /index.html)? (y/N) Y

Figura 10.4: Finalizando a configuração

Concluímos a configuração para utilizar o serviço de hospedagem do Firebase.

Agora, podemos implantar o projeto informando o comando no terminal:

```
firebase deploy only --hosting
```

Repare que já utilizamos o comando `firebase deploy` no capítulo anterior para enviar as funções ao servidor. A diferença está no acréscimo da opção `only --hosting`, indicando para subir somente os arquivos para hospedagem do projeto.

A imagem a seguir apresenta o resultado do console, com o deploy completo.

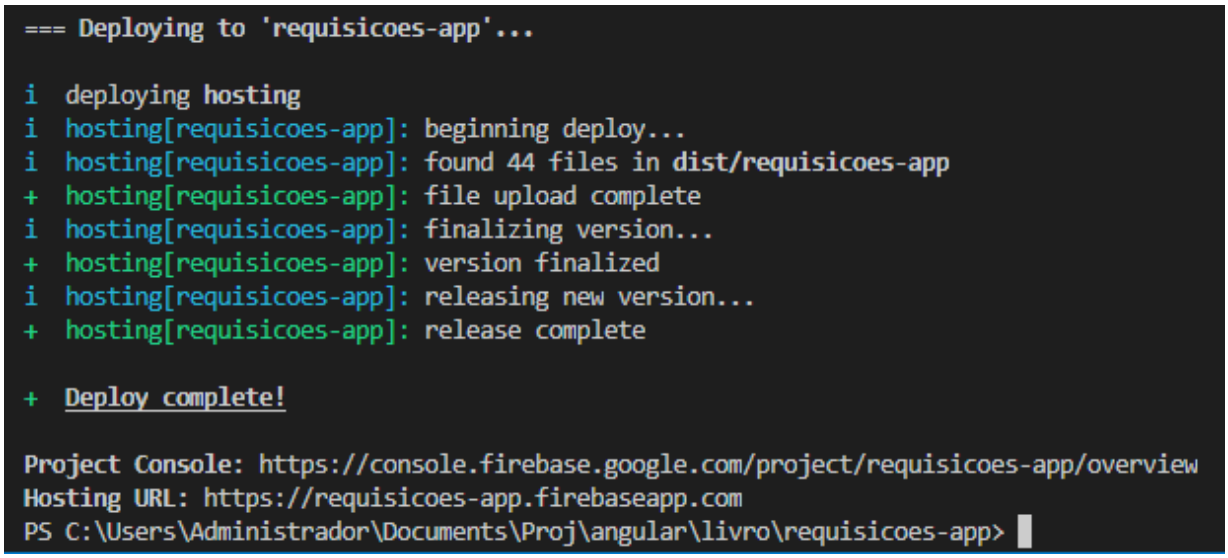


Figura 10.5: Pasta com os projeto para produção

A informação *Hosting URL* está o endereço da aplicação, `https://requisicoes-app.firebaseapp.com`, conforme imagem a seguir do navegador:

https://requisicoes-app.firebaseapp.com/login
LG Optimus...
384
640
125%
Mobile
Sistema de Requisições
Email
teste@teste.com
Senha
Senha
LOGAR
OU
RECUPERAR

Figura 10.6: Aplicação hospedada no Firebase

Toda vez que desejarmos implantar uma versão atualizada da aplicação será necessário somente construir o projeto e realizar o deploy.

No painel do Firebase (https://console.firebase.google.com) podemos acessar o histórico de implantações da aplicação, reverter um lançamento, além de configurar um domínio próprio.

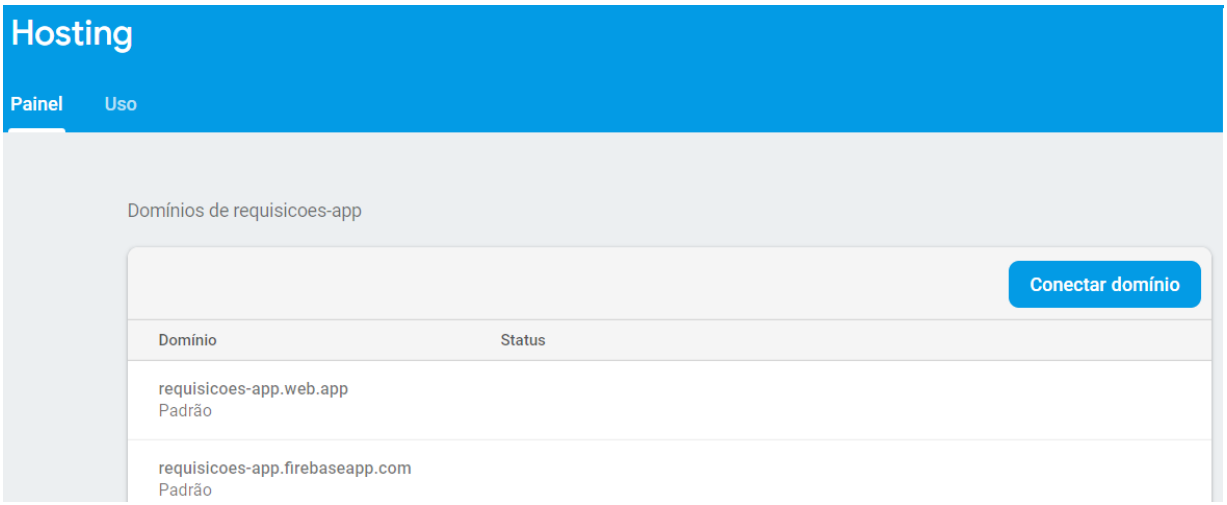


Figura 10.7: Painel do Hosting no Firebase

E assim, com poucos passos, concluímos efetivamente a integração do projeto com mais um serviço da plataforma Firebase, hospedando a aplicação em um ambiente seguro e com conteúdo por SSL, além de poder utilizar os subdomínios `web.app` e `firebaseapp.com`.

10.2 IVY - O novo compilador do Angular

A versão 8 do angular introduziu um novo compilador do framework, o **Ivy**.

Ivy

Trata-se de uma reescrita completa do compilador e tempo de execução sem alterar a forma como escrevemos aplicações em Angular.

Entre os objetivos dessa mudança, podemos listar:

- Melhor tempo de compilação (com uma compilação mais incremental);
- Redução do tamanho de compilação;
- Carregamento lento de componente em vez de módulos;
- Sistema novo de detecção de alterações não baseado em `zone.js` (utilizado para detectar quando determinadas operações assíncronas ocorrem para acionar um ciclo de detecção de alterações. Um exemplo do uso é quando o Angular realiza verificação e execução de atualizações da interface do usuário.).

Podemos definir o uso do Ivy na construção de um novo projeto.

No terminal, informe:

```
ng new meu-projeto  --enable-ivy
```

A flag `--enable-ivy` já configura o projeto, definindo as propriedades necessárias para uso do novo compilador.

Para um projeto existente, as seguintes alterações devem ser realizadas:

1. Encontre o arquivo `tsconfig.app.json` na raíz do projeto e defina a propriedade `enableIvy` para `true` dentro de `angularCompilerOptions` .

   ```
   {
   "compilerOptions": { ... },
   "angularCompilerOptions": {
   ```

```
    "enableIvy": true
  }
  }
```

2. No arquivo de configuração `angular.json`, defina as opções de construção padrão para o seu projeto para sempre usar a compilação `aot`.

> **AOT**
>
> O compilador Angular *Ahead of Time* (AOT) converte o código Angular HTML e TypeScript em código JavaScript eficiente durante a fase de compilação, antes que o navegador faça o download e execute esse código, fornecendo uma renderização mais rápida.

```
{
  "projects": {
    "my-existing-project": {
      "architect": {
        "build": {
          "options": {
            ...
            "aot": true,
          }
        }
      }
    }
  }
}
```

Está feito! O projeto está configurado para utilizar o novo compilador do Angular.

10.3 Considerações finais

Chegamos ao final do livro, mas não do desenvolvimento.

Encontrar, filtrar, selecionar e roteirizar a quantidade de materiais disponíveis hoje na Web é quase uma tarefa hercúlea.

O programador iniciante tende a ficar perdido nessa imensidão de conteúdos, levando tempo e muitas vezes desânimo na construção de um sistema real.

Dessa forma, o objetivo deste livro foi apresentar as principais características do framework Angular integrando com a plataforma Firebase, desenvolvendo um sistema de requisições.

Desenvolvemos requisitos que são comuns a todo sistema como Login, Área Restrita, Menu e CRUDs de entidades, explorando conceitos chaves da linguagem como Componentes, Módulos, Rotas e Serviços.

Integramos a aplicação a diversos recursos do Firebase como autenticação, banco de Dados, armazenamento de arquivos, execução de funções nas nuvens e hospedagem.

Assim, o leitor pode utilizar essa obra como referência ou ponto de partida para a construção de uma gama de sistemas, adequando aos seus requisitos e necessidades da implementação.

Espero ter contribuído para que mais desenvolvedores utilizem esse incrível framework na construção de soluções.

Lembre-se, você não precisa ter talento, mas ser apaixonadamente curioso ou curiosa. (Einstein)

Obrigado!

10.4 Links consultados

Documentação do Angular - https://angular.io/docs

Documentação do Firebase - https://firebase.google.com/docs?hl=pt-BR

AngularFire - https://github.com/angular/angularfire2

Cloud Firestore - https://firebase.google.com/docs/firestore

Cloud Sorage - https://firebase.google.com/docs/storage

Cloud Functions - https://firebase.google.com/docs/functions

Hosting - https://firebase.google.com/docs/hosting

PrimeNG - https://www.primefaces.org/primeng

Nodemailer - https://nodemailer.com/about

Medium - https://medium.com/@kheronn.machado

GitHub - https://github.com/kheronn